DQ-lang Project

Language, Libraries and Tools

Contents

1. DQ-lang Project 4
1. DQ-lang Project Pages 4
1.1. DQ-lang Project Introduction 5
1.2. Features and Design Goals 7
1.3. Getting Started 11
2. DQ Programming Language 14
2.1. Basic Syntax 15
2.2. Basic Types and Conversions 17
2.3. Variables, Constants, References, and Scope 20
2.4. Expressions and Operators 23
2.5. Literals and Integer Promotion 32
2.6. Statements and Control Flow 37
2.7. Functions 41
2.8. Character and String Types 46
2.9. String Functions 52
2.10. Text Formatting 67
2.11. Arrays and Slices 76
2.12. Enumerations 86
2.13. Structures and Unions 88
2.14. Pointers 93
2.15. Objects and Properties 105
2.16. Memory Management and Autofree 125
2.17. Exceptions and Error Handling 132
2.18. anyvalue 138
2.19. Modules and Packages 142
2.20. Attributes 147
2.21. Compiler Directives 151
2.22. C Interoperability 156
2.23. Assembly Functions 160
3. DQ Compiler and Tools 164
3.1. Compiler and Tool Overview 165
3.2. Runtime and Program Execution 167
3.3. Linux Development Environment 169
3.4. Windows Development Environment 172
3.5. Compiler Usage and Options 176

3.6. Project Files (`.dqproj`) 178
3.7. Build Process and Module Artifacts 188
3.8. Targets and Cross-Compilation 190
3.9. VSCode Setup and Debugging 192
4. Standard Packages 195
4.1. Library Overview 196
4.2. `sys` 197
4.3. `print` 199
4.4. `file` 200
4.5. `json` 202
4.6. `threads` 204
4.7. `time` 210
4.8. `nanonet` 214
4.9. `shell` 216
4.10. Additional Standard Packages 221
5. Appendices 222
5.1. Example Programs 222
5.2. Building the Compiler from Source 224
5.3. Contributor and Implementation Notes 227
5.4. Module and Interface Internals 230
5.5. Exceptions and Backtraces 232
5.6. Release and Cross-Build Notes 234
5.7. Glossary and Index 239
5.8. Documentation Coverage 241

1. DQ-lang Project

1. DQ-lang Project Pages

Contents:

- Introduction
- Features
- Getting Started

1.1. DQ-lang Project Introduction

DQ is a **general-purpose** and **human-friendly** programming language.

General-purpose, because it is intended to cover many use cases, including scripts and tools, server applications, embedded applications, GUI applications, and games.

Human-friendly, because DQ syntax and library interfaces are designed to make program source code easy to read and write for people.

The DQ-lang Project is being developed to offer a serious alternative to Python and C++. The DQ language mostly uses established concepts from other programming languages. Its syntax combines elements of Pascal, Python, and C/C++. The name reflects these influences: D is the letter after C, and Q is the letter after P.

DQ source files (.dq files) are translated into directly executable machine code, so DQ applications run very fast, matching the speed of programs written in C++ or Rust. For machine code generation, the DQ compiler uses the well-tested LLVM framework, which also supports different architectures and targets, such as x86, ARM32, ARM64, ARM Cortex-M, RISC-V, and WebAssembly (WASM). The DQ compiler allows you to create applications for Windows, Linux, and bare-metal targets, including through cross-compilation.

1.1.1 DQ Language Self-Introduction

```
use print

func *Main():
    PrintLn('DQ Short Introduction')

    if 3 / 2 == 1.5  and 3 div 2 == 1  and uint32(3) - uint32(5) < 0:
        PrintLn('DQ is human friendly.')
    endif

    var sarr : [5]int = [2, 3, 5, 7, 11]
    Print('Static array:')
    for i : int = 0 count sarr.length:
        Print(' {}', [sarr[i]])
    endfor
    PrintLn()

    var darr : [*]float = [2, 3.14, 7 / 3]
    darr.Append(13 / 2)
    darr.PopFirst()
    Print('Dynamic array:')
    for f :? in darr:
        Print(' {}', [f])
    endfor
    PrintLn()

    var ds : str = 'This is a'
    ds.Add(' dynamic string.')
    var dpos : int = ds.IndexOf('dynamic')
    ds.Insert(dpos, 'heap allocated ')
    PrintLn(ds)

    PrintLn('String slicing: ds[2:6]="{}", ds[$end-5:]="{}"',
        [ds[2:6], ds[$end-5:]] )

    var x : int = 7
    PrintLn('original x = {}', [x])
    var xptr : ^int = %x   // %x = address of x
    ref xref = x
    xref = 42
    if xptr^ == x  and xref == x:
        PrintLn('Supports pointers and references. x = {}', [x])
    endif

    PrintLn('DQ has many more features. See further examples.');
endfunc
```

Source: https://github.com/nvitya/dq-lang/blob/main/examples/dq_intro_1.dq

This example covers only a small subset of the capabilities of the DQ language. One very important feature is support for **objects with inheritance and virtual functions**. For a quick example, see A Short Object Demo in the Objects and Properties chapter.

For a more complete list of features, read Features and Goals.

For a complete description of the DQ language, see the DQ Language Reference.

1.1.2 Getting Started

To get the DQ compiler and start experimenting with the language, follow the instructions on the Getting Started page.

1.1.3 Project State

Compiled DQ programs for Windows and Linux use the standard, well-tested C library for low-level system functions. With the reliable LLVM code generator and extensive compiler testing, the DQ compiler and its produced applications are already fairly stable.

During the development of DQ applications, bugs and language defects have been detected and fixed. AI assistance is used - under strict supervision - in compiler development, making it possible to fix bugs or implement changes very quickly.

Using DQ in critical applications is not yet recommended because it has not undergone sufficient field testing. (The DQ-lang project started in January 2026 and became ready for application development only around September 2026.) **Every new project must start somewhere**, and building a supporting community takes time.

You can already use DQ for less critical applications and tools, and I recommend doing so.

The DQ standard packages offer user-friendly, object-based interfaces for several programming tasks, including database handling, threading, HTTP serving, and JSON handling. Since DQ is a new language (2026), library support is still incomplete. DQ has a very flexible module system and a simple way to import functions from external C-compatible libraries, which opens unlimited possibilities for application development.

1.1.4 Project Page

The DQ-lang project is available on GitHub at:

github.com/nvitya/dq-lang

1.1.5 Discussions

For questions, feedback, language-design discussions, examples, and general conversation about the DQ programming language and compiler, you can use the discussion page on GitHub:

https://github.com/nvitya/dq-lang/discussions

You can also use the DQ community subreddit:

https://www.reddit.com/r/dqlang/

For confirmed bugs and concrete feature requests, please use the GitHub issue tracker.

1.1.6 Supporting the DQ-lang Project

Please support the DQ-lang project by trying it out, writing and sharing DQ applications, and giving feedback!

1.1.7 Further Reading

- Language Reference
- Compiler and Tools
- Standard Packages

1.2. Features and Design Goals

DQ aims to combine the convenience of a language used for everyday tools with static type checking, native compilation, fast execution and direct access to the machine. The goal is to make programs easy to read and maintain as they grow from small utilities into larger applications.

Most of DQ's individual features are established ideas from other programming languages. What distinguishes the language is the combination: readable syntax, deliberate expression rules, objects with inheritance, automatic cleanup, exceptions, and low-level control within one language.

1.2.1 Motivation

DQ grew out of experience with Pascal application development, C++ embedded software, and a large Python project for device production testing. The motivation was practical: keep the convenience of writing small programs, catch more mistakes before running them, and retain the ability to work with hardware and existing native libraries.

Those needs connect application programming and systems programming. A tool might start by reading a configuration file, then grow to communicate with a device, store measurements in a database, and serve a web interface. DQ's goal is to support that growth without requiring a different language for the parts that need native performance or direct memory access.

1.2.2 Familiar Ideas, Deliberate Choices

DQ draws on four main language traditions, combining features that support human-readable code, predictable calculations, and direct control of the machine.

These ideas are adapted to work together under DQ's own rules. Human readability guides the syntax, while explicit types and storage choices keep the program's behavior understandable.

1.2.2.1 Features Adopted from Python

- Human-readable source code.
- Readable blocks introduced by `:`. DQ adds explicit closers such as `endif` and `endfunc`, preserving the visual structure while making indentation optional.
- Statements on separate lines without required semicolons, keeping the source easy to read with little punctuation.
- Flexible module system, with non-merging, merging imports and re-exports
- Excellent, user-friendly, modern APIs
- String, array slicing

1.2.2.2 Features Adopted from Pascal / Delphi

- Human-readable source code.
- Strict type-safety.
- Implicit function `result`
- Floating-point division with `/`, with a separate operator for integer division.
- Explicit rounding when converting floating-point values to integers.
- Boolean values distinct from numbers, requiring explicit comparisons in conditions.
- Object properties, single inheritance, and exception handling.
- Module interface/implementation split
- Elegant variable arguments and anyvalue (variants)
- Function references, object method references
- Simple and fast build system

1.2.2.3 Features Adopted from C / C++

- Compact expression syntaxes.
- Pointers, structures, and unions for direct control over memory and data layout.
- Embeddable objects.
- Fixed-width numeric types and bitwise operations for programming close to the machine.
- Direct access to C-compatible libraries.
- C-style comments and conditional-compilation directives.
- Text formatting
- Efficient data processing.

1.2.2.4 Features Adopted from JavaScript

- String and array properties accessed directly on the value, such as `text.length`.
- String and array methods called on the value, such as `text.IndexOf("word")` and `text.StartsWith("prefix")`, keeping text operations easy to discover and read.

See String Functions for DQ's string properties and methods.

1.2.3 Some Distinctive DQ Design Choices

- **Carefully selected operator precedence:** The precedence rules reduce the need for parentheses in common expressions, like this: `flags & mask <> 0 and count > 0`
- **Separate logical and bitwise operators:** `and`, `or`, and `not` express Boolean logic; `&`, `|`, `~`, and `xor` express bit operations. The distinct notation makes the intended operation easy to recognize.
- **Explicit special functions:** `*` marks functions with language-defined roles, such as `*Main()`, `*Create()`, and `*Destroy()`. The marker makes those roles visible and lets the compiler reject an unrecognized special function name, including a misspelling.
- **Explicit object embedding:** `<-` constructs an embedded object whose lifetime follows its containing scope or object. For example, `var file <- OFile()` makes storage and automatic cleanup visible in the declaration.
- **Embedded string:** `embstr(n)` for simple, embedded-friendly, heap-less string handling
- **Conditional compilation aware of program symbols:** Directives such as `#if` and `#ifdef` can use visible program symbols as well as compile-time defines. For example, `#ifdef somelocalvar` checks whether a local variable exists in the current scope.
- **Customizable runtime:** DQ packages and runtime libraries are distributed as source code and compiled as part of the application build. Compile-time defines can control their conditional compilation, allowing supported features and settings to be tailored to the application and target.

1.2.4 Readable Code with Explicit Structure

DQ favors syntax that makes the purpose of a statement visible. Types follow names, function return types follow `->`, and blocks end with words such as `endif` and `endfunc`. Indentation is recommended but does not define the blocks. Semicolons are unnecessary when statements are on separate lines.

```
func Clamp(value : float, lower : float, upper : float) -> float:
    if value < lower:
        return lower
    elif value > upper:
        return upper
    endif
    return value
endfunc
```

Naming conventions distinguish functions such as `ReadSensor` from values such as `temperature`. Object methods can access their members without repeating a `self.` prefix. The intention is to keep frequently used code easy to scan, while retaining clear declarations and boundaries.

See Basic Syntax and Functions.

1.2.5 Expressions That Say What They Mean

Small differences in arithmetic and conversion rules can prevent common mistakes. DQ makes several explicit choices:

- `/` performs floating-point division: `3 / 2` is `1.5`. Use `div` when integer division is intended.
- Conditions require Boolean values. A number or pointer does not implicitly become `true` or `false`; write a comparison such as `count <> 0`.
- Floating-point values do not implicitly become integers. Choose the rounding behavior with `Round`, `Floor`, `Ceil`, or `Trunc`.
- Assignment is a statement, so an accidental `=` cannot act as a comparison inside a condition.
- Logical operations use `and`, `or`, and `not`, separately from bitwise `&`, `|` and `~` operations.

For a C or C++ programmer, these rules change some familiar defaults. For a Python programmer, floating-point division is familiar, while explicit Boolean conditions are a stricter requirement. The goal is to make the intended operation apparent in the source.

See Expressions and Operators.

1.2.6 Convenient Data with Control over Storage

Strings, dynamic arrays, slices, and formatted text are part of everyday DQ programming. They do not require writing allocation and resizing code for each use. Fixed arrays, structures, embedded objects, and fixed-storage strings are also available when storage size and placement matter.

DQ combines automatic storage management with explicit ownership. Strings and dynamic arrays manage their storage; embedded objects are destroyed with their containing scope or object. Autofree owners release heap allocations when they are replaced or leave scope, including during exception handling.

This provides deterministic cleanup without a tracing garbage collector. Compared with a garbage-collected object model, the programmer has more control over when resources are released, but also more responsibility: borrowed references must not outlive their storage, and manually allocated objects need an owner or an explicit release. DQ's raw pointers and manual allocation do not provide a general memory-safety guarantee.

See Memory Management and Autofree.

1.2.7 Objects and Error Handling for Application Code

DQ supports single inheritance, virtual methods, constructors, destructors, and properties. These let a program express related kinds of objects and expose simple interfaces while keeping implementation details inside the object. They coexist with plain functions and structures, so object-oriented design can be used where it helps.

Exceptions keep the normal sequence of operations readable. A function can open a file and write several values without repeating an error-result check after every operation. Failures reach a matching handler, and an unhandled exception normally produces a message and backtrace before the program exits. Automatic cleanup and `finally` blocks handle resources along the way.

On supported hosted targets, unwind tables provide zero-cost exception handling: normal calls do not need an extra exception check. Throwing and handling an exception still involves runtime work.

See Objects and Properties and Exceptions and Error Handling.

1.2.8 Modules and a Short Build Workflow

A DQ module is a source file with a public interface and an optional private `implementation` section. Imports can expose selected names directly or require namespace qualification. This gives growing programs explicit boundaries without maintaining separate C-style header files.

For a small program, `dq-run app.dq` compiles and runs it. The compiler follows module dependencies and reuses compiled artifacts, so a basic DQ program does not need a separate build script. Project files provide configuration as the build grows.

The aim is a short edit–run cycle with the checks of a compiled language. Compilation time and execution speed depend on the program and build settings; using LLVM is a foundation for optimization, not a guarantee about benchmarks.

See Modules and Packages and Compiler and Tools.

1.2.9 Native Libraries and Embedded Targets

DQ can call C-compatible functions, including variadic functions, and link to existing native libraries. Pointers, explicit-width numeric types, structures, unions, and assembly functions support work close to the hardware.

The compiler uses LLVM for code generation and supports hosted and bare-metal targets. The same language can serve application and embedded development, but the available runtime facilities differ: a bare-metal program needs its own platform support, and not every target supports exceptions or every package.

See C Interoperability and Targets and Cross-Compilation.

1.2.10 Goals and Current Scope

DQ's direction is to make ordinary programming tasks straightforward while keeping performance, storage, and resource lifetimes understandable. Readable source, useful diagnostics, consistent library interfaces, and a manageable build process are all part of that goal.

Language features are only part of choosing a tool. Existing libraries, developer experience, platform support, and long-term maintenance matter too. DQ is a young project, and its ecosystem and field experience are still growing. For its current development state, see the Project Introduction; for available libraries, see Standard Packages.

1.3. Getting Started

This chapter takes you from installing DQ to running your first program. You need the DQ compiler and a text editor. Visual Studio Code is recommended for syntax highlighting, compiler diagnostics, and running programs from the editor.

1.3.1 Install DQ

Download a compiler package for your operating system from the DQ releases page. Choose a compiler archive from the release assets; the GitHub **Source code** archives contain the project sources rather than ready-to-run compiler tools.

Follow the setup guide for your system, then return here:

- Linux Development Environment: installing a release, setting up your shell and editor, and adding external libraries.
- Windows Development Environment: extracting the Windows package, configuring `PATH`, setting up the editor, and adding external libraries.

The two main tools are:

Tool	Purpose
`dq-comp`	Compiles DQ source files and links them into an executable.
`dq-run`	Compiles a DQ program and immediately runs it.

In a terminal with your DQ environment configured, check that the compiler is available:

```
dq-comp --version
```

The command should print the installed compiler version.

1.3.2 Write Your First Program

Create a directory for your experiments. In that directory, use your editor to create a plain text file named `hello.dq`. The `.dq` extension identifies DQ source code. On Windows, make sure the file is not accidentally saved as `hello.dq.txt`.

Put this code in the file and save it:

```
use print

func *Main():
    PrintLn("Hello from DQ!")
endfunc
```

The program has three main parts:

- `use print` makes the standard package's printing functions available.
- `func *Main():` begins the program's entry function. Execution starts here after the runtime has initialized the program.
- `PrintLn(...)` writes text to the terminal and ends the output line. `endfunc` closes the function.

Text inside double quotes is a string: a sequence of characters representing text. The indentation makes the function body easier to read.

1.3.3 Compile and Run

Open a terminal in the directory containing `hello.dq` and run:

```
dq-run hello.dq
```

The program prints:

```
Hello from DQ!
```

`dq-run` invokes the compiler, creates an executable, and starts it. If compilation reports an error, correct the indicated line in your source file, save it, and run the command again.

You can also compile and run as separate steps. On Linux:

```
dq-comp -g -O0 hello.dq
./hello
```

On Windows, in Command Prompt or PowerShell:

```
dq-comp -g -O0 hello.dq
.\hello.exe
```

The compiler produces `hello` on Linux and `hello.exe` on Windows. The `-g` option includes debugging information, and `-O0` disables optimization so the generated code is easier to debug. When called without compiler options, `dq-run` uses both options automatically.

1.3.4 Print Values

Formatting lets you insert values into a message. Each `{}` is replaced by the next value in the list between `[` and `]`:

```
use print

func *Main():
    var name : str = "DQ"
    var count : int = 3
    PrintLn("Hello, {}! Count = {}", [name, count])
endfunc
```

Here, `var` declares a variable: a named place to store a value. `str` is a string type, and `int` is an integer type for whole numbers. This program prints `Hello, DQ! Count = 3`.

1.3.5 Program Arguments and Exit Status

Programs can receive arguments from the command line. Save this example as `args.dq`:

```
use print

func *Main():
    for i : int = 1 count ArgCount() - 1:
        PrintLn("Argument {}: {}", [i, ArgStr(i)])
    endfor
endfunc
```

Run it with two arguments:

```
dq-run args.dq -- one two
```

The output is:

```
Argument 1: one
Argument 2: two
```

`ArgCount()` includes the program name at index `0`, so this example starts at index `1`. `ArgStr(i)` reads the argument at that index. These functions are available through the implicit `sys` runtime interface.

Compiler options go before the source filename. Program arguments go after it; the optional `--` separator makes the boundary explicit.

`*Main` has no return value. The program normally exits with status `0`, which means success. Set the global `exit_code` variable to another value before leaving `*Main` to report failure, for example `exit_code = 1`.

1.3.6 Continue Learning

Read the DQ Programming Language overview for the language topics in book order. The Standard Packages provide printing, file access, JSON handling, networking, and other services.

For build options and larger programs, continue with Compiler Usage and Options and Project Files.

2. DQ Programming Language

This chapter is the complete DQ language reference, organized from basic syntax to platform access. Each topic introduces its concepts, then describes syntax, type rules, lifetime constraints, and examples in the same chapter.

If DQ is new to you, begin with Getting Started to install the compiler and run a program. This reference describes DQ 0.93.16. DQ is still under development, so match the documentation to your compiler version.

Most code blocks illustrate declarations or statements in their surrounding context; examples containing `func *Main()` are complete programs unless they explicitly import companion files. Copy a complete program into a `.dq` file and run it with `dq-run filename.dq`. Assembly examples identify their target.

Contents

- Basic Syntax
- Basic Types and Conversions
- Variables, Constants, References, and Scope
- Expressions and Operators
- Literals and Integer Promotion
- Statements and Control Flow
- Functions
- Character and String Types
- String Functions
- Text Formatting
- Arrays and Slices
- Enumerations
- Structures and Unions
- Pointers
- Objects and Properties
- Memory Management and Autofree
- Exceptions and Error Handling
- anyvalue
- Modules and Packages
- Attributes
- Compiler Directives
- C Interoperability
- Assembly Functions

2.1. Basic Syntax

A DQ program is text stored in one or more `.dq` files. Each file is a module. A function groups statements that perform a task; `*Main` is the entry function of an ordinary application. For installation and the compile/run commands, see Getting Started.

```
use print

func *Main():
    // A variable stores a value under a name.
    var count : int = 3
    PrintLn("Count = {}", [count])
endfunc
```

`use print` imports printing functions. The colon starts a function body and `endfunc` closes it. A newline normally finishes a statement.

2.1.1 Source Files and Names

Source text can contain Unicode text in strings and comments. Names are case-sensitive: `count`, `Count`, and `COUNT` are different identifiers. Identifiers start with an ASCII letter or `_` and continue with ASCII letters, digits, or `_`. Unicode identifiers are not supported.

Keywords are lowercase, for example `func`, `var`, `if`, and `endfunc`. Built-in names such as `SizeOf`, `Ord`, `Round`, and `Object` keep their documented case. Use descriptive names; the language does not require a particular naming prefix.

A `.dq` file is a source module. `.dqh` files conventionally hold public header fragments and `.dqi` files hold other included fragments. Textual includes are explained in Compiler Directives; module imports are explained in Modules and Packages.

2.1.2 Statements, Whitespace, and Blocks

Spaces and tabs separate tokens. Indentation makes nesting clear, but does not create or close blocks. A semicolon explicitly ends a statement and is needed between multiple statements on the same physical line:

```
var x : int = 1
var y : int = 2; x += y
```

Expressions can continue across lines inside balanced parentheses or brackets. Use the matching closer for each colon block: `endif`, `endwhile`, `endfor`, `endfunc`, `endobject`, `endstruct`, `endunion`, or `endtry` as appropriate. Supported constructs also allow braces:

```
func Twice(value : int) -> int {
    return value * 2
}
```

A block must consistently use its chosen closing style. Assembly function bodies require the colon/`endfunc` style. This book normally uses named closers.

2.1.3 Comments

`//` starts a comment through the end of the line. `/*` and `*/` enclose a block comment. Comments act as whitespace; comment markers inside a string are text.

```
// A line comment.
var count : int = 3 /* a comment between tokens */
```

2.1.4 Literals

A literal writes a value directly in source code:

Kind	Examples
Whole number	`42`, `0xFF`
Floating point	`3.14`, `2e3`
Boolean	`true`, `false`
Empty reference/pointer	`nil`
Text	`"hello"`, `'hello'`, `"/"`
Character	`'/'`, `'€'`
Array	`[1, 2, 3]`
Default aggregate	`{}`

A leading minus is an operator: `-42` negates the literal `42`. Numeric literals receive the type required by their context; there is no need to add `.0` merely to initialize a floating-point variable with a whole number.

Double quotes always denote text. Single quotes containing exactly one Unicode scalar denote a `wchar`; single quotes containing zero or multiple scalars denote text. Consequently, use `"/"` when comparing a string with one-character text.

Common escapes are `\n`, `\r`, `\t`, `\\`, `\"`, and `\'`. `\xNN` inserts a byte using exactly two hexadecimal digits. Triple-quoted text spans lines:

```
var message : str = """
    first line
      indented line
    """
```

You can use either triple double quotes or triple single quotes. The text has no extra empty line at the start or end. DQ removes the spaces used to line up the text with the closing quotes and keeps any extra indentation. Escape sequences still work. See Multiline Text for examples showing the resulting text. For character and string representations, see Character and String Types. Byte and Unicode operations are described in String Functions.

2.1.5 Declarations and Expressions

The usual declaration forms are `var name : Type = value`, `const NAME : Type = value`, and `type Name = Type`. A colon introduces a type annotation in a declaration. `=` assigns a value; `==` compares values. Assignment is a statement without resulting a value. Assignment is not chainable so `a = b = 1` is invalid (compiler error).

An expression computes a value, such as `2 + 3` or `count > 0`. Conditions require `bool`: write `count <> 0`, using a number (or anything else than `bool`) as a condition is not allowed. The inequality operator is `<>`; `!=` is not accepted.

Read more about operators and expressions in the chapter Expressions and Operators

2.1.6 Attributes and Directives

`[[...]]` attaches attributes to declarations. `#` starts compiler directives, including conditional compilation and source inclusion. They have different roles from executable statements; see Attributes and Compiler Directives.

2.2. Basic Types and Conversions

A type describes the values a variable can store and the operations allowed on it. DQ checks types during compilation. It distinguishes numbers, Boolean values, characters, text, pointers, and compound types rather than converting freely between unrelated categories.

```
use print

func *Main():
    var count : int = 3
    var ratio : float = count / 2
    var ready : bool = count > 0
    PrintLn("{} {} {}", [count, ratio, ready])
endfunc
```

2.2.1 Scalar Types

Type	Values and representation
`bool`	`true` or `false`; not an integer.
`int`, `uint` (1)	Signed/unsigned integers as wide as a pointer.
`int8`, `int16`, `int32`, `int64` (1)	Signed integers of the specified bit width.
`uint8`, `uint16`, `uint32`, `uint64` (1)	Unsigned integers of the specified bit width.
`byte`	Alias for `uint8`. Preferred instead of `uint8`
`float32`, `float64`	Floating-point numbers of the specified width.
`float` (2)	Floating-point type chosen by the target. `float32` on the single-precision FPU targets arm_m4f-bare, arm_m33f-bare, and arm_m7f-bare. It is `float64` on other current targets, including arm_m7fd-bare and WebAssembly.
`char`	One unsigned 8-bit byte or UTF-8 code unit.
`char16`	One unsigned 16-bit UTF-16 code unit.
`wchar`	One unsigned 32-bit Unicode scalar value.
`pointer`	Generic address with no pointee type.

Notes:

(1): For an unsigned integer with `N` bits, the range is `0` through `2^N - 1`.
For a signed integer, it is `-2^(N-1)` through `2^(N-1) - 1`.
`int` / `uint` are 32 bits on 32-bit targets and 64 bits on 64-bit targets.
Choose a fixed-width type when a file format, protocol, or external ABI requires it.

(2): Use explicit floating-point widths avoid making a binary interface depend on the target default.

2.2.2 Other Type Categories

Form	Purpose	Chapter
`str`, `rostr`, `strslice`, `embstr(N)`	Text with different storage and borrowing rules.	Character and String Types
`[N]T`, `[*]T`, `[]T`	Fixed arrays, dynamic arrays, borrowed array views.	Arrays
`enum`	A distinct set of named values.	Enumerations
`struct`, `union`	Direct field storage or overlapping field storage.	Structures and Unions
`^T`	An address pointing to a `T`.	Pointers
`object`, `Object`	Object reference types; `Object` is the common object base type.	Objects
`func(...)`	A function reference with a particular signature.	Functions
`anyvalue`	A boxed scalar/text value for heterogeneous APIs.	anyvalue

`autofree T` qualifies object-reference and pointer types with automatic cleanup; it does not change the underlying pointer representation. See Memory Management and Autofree.

2.2.3 Type Aliases

`type` introduces another name for an existing type:

```
type TCount = uint32
type FTransform = func(value : int) -> int
```

An alias does not create a new, incompatible type. In contrast, two separately declared enumerations, structures, or unions remain distinct even if their contents look identical.

2.2.4 Numeric Conversions

Whole-number literals are contextual. Write `var scale : float = 2` or `var size : uint8 = 3` directly. Integer expressions can convert to another integer storage width; narrowing keeps the low bits. Integer-to-floating-point conversion is available where the expected numeric type requires it.

```
var small : uint8 = 250
small = small + 10       // 260 is stored as 4 in eight bits
var real : float64 = small
```

A destination does not retroactively widen the operands of an expression. The literal typing, calculation width, and signedness rules are explained in detail in Literals and Integer Promotion.

Floating-point values do not implicitly convert to integers. Use `Round`, `Floor`, `Ceil`, or `Trunc` to choose the rounding operation:

```
var value : float = 3 / 2
var rounded : int = Round(value)
var downward : int = Floor(value)
var upward : int = Ceil(value)
var truncated : int = Trunc(value)
```

Type-call syntax such as `uint16(value)` requests a defined explicit conversion. The `expression as Type` spelling uses the same conversion rules. A cast is not a substitute for a rounding operation, range check, or valid memory address.

2.2.5 Boolean Isolation

Numbers and Boolean values are separate. Conditions and logical operators take `bool`; `if count:` and `int(true)` are not numeric conversion shortcuts. Convert through explicit program logic, for example `iif(ready, 1, 0)` or `n <> 0`.

2.2.6 Characters, Enumerations, and Pointers

Character types are distinct from integers. Prefer the read-only `ch.ord` property to obtain a character code as `int`; `Ord(ch)` is equivalent. `char(code)`, `char16(code)`, and `wchar(code)` request explicit conversions. Checked `IntToChar` / `IntToWchar` helpers provide raising and fallback forms. A constant character literal below 256 can initialize a `char` directly. Perform character arithmetic on its integer code, for example `char(ch.ord + 1)`.

Enums use `Ord`, `.ord`, `FromOrd`, and `TryFromOrd`; ordinary integer casts do not bypass enum membership checks. See Enumerations.

Integer/address and unrelated pointer conversions require explicit casts. Converting an address does not establish its lifetime, alignment, or ownership.

2.2.7 Size and Layout Queries

`SizeOf(Type)` and `SizeOf(value)` report storage size in bytes. `OffsetOf(Type, field)` reports a structure or union member's byte offset. Use these queries rather than assuming pointer widths or padding. `Len(value)` and `.length` measure logical array elements or text bytes, not the total storage size.

2.3. Variables, Constants, References, and Scope

A variable names storage whose value can change. A constant names a value known during compilation. A reference gives another name to existing storage. A scope controls where a declaration can be found.

```
use print

const LIMIT : int = 10

func *Main():
    var count : int = 2
    ref alias = count
    alias += 3
    PrintLn("{} / {}", [count, LIMIT]) // 5 / 10
endfunc
```

2.3.1 Variables and Initialization

Declare a variable with `var name : Type`, normally followed by `= initializer`. An assignment replaces its current value:

```
var count : int = 0
count = 3
count += 1
```

Initialize scalar locals before reading them. A declaration such as `var count : int` reserves storage but does not make its value readable; the compiler tracks definite assignment across branches and reports uninitialized uses. A function result must likewise be assigned on every normal exit path.

Sized embedded strings are initialized automatically: `var s : embstr(32)` starts empty and can be read or appended to immediately. See Embedded Strings.

Global storage is zero/default-initialized before program startup. Types with managed lifetimes, including dynamic arrays and autofree references, have their own empty/default initialization and cleanup rules. Use an explicit initializer when the intended value matters. `{}` explicitly zero/default-initializes a supported aggregate such as a structure or fixed array.

Object fixed storage uses `<-` rather than `=`; see Objects and Properties.

2.3.2 Restricted Type Inference

Write `?` in place of a declared type when an initializer independently supplies an eligible concrete typed pointer, structure, or object-reference type:

```
struct SPoint:
    x : int
    y : int
endstruct

var point : SPoint = {10, 20}
var copy : ? = point
var address : ? = %point
```

Inference requires `=` and preserves the initializer's static type. Numeric, Boolean, text, enum, raw `pointer`, and other unlisted types are not inferred. In particular, `var n : ? = 3` and `var p : ? = nil` are errors. The separate array notation `[?]T` infers a fixed array's length, not its element type.

For local variables initialized by `new` or `TryCast(...)`, the type annotation can be omitted entirely. For example, `var child = TryCast(OChild, base)` infers `OChild` from the first argument and creates an ordinary reference without `autofree`. See Checked Object Casts.

`new autofree T` provides a shorthand for declaring an autofree variable; ordinary `?` inference does not add `autofree`. See Memory Management and Autofree.

2.3.3 Local References

`ref name = existing_value` binds an alias to addressable compatible storage. Writes through the alias update the original storage. The binding must remain valid throughout the alias's use. A reference is not a nullable pointer value.

A reference cannot extend the lifetime of a local, a temporary, an array element, or a property result. Function reference parameter modes are described in Functions.

2.3.4 Constants

`const` requires a compile-time constant initializer:

```
const BUFFER_SIZE : int = 4096
const HALF_BUFFER : int = BUFFER_SIZE div 2
```

Related constants can share a declared type:

```
const(uint32):
    FLASH_BASE = 0x08000000
    FLASH_END  = FLASH_BASE + 0x7FFF
    SRAM_BASE  = 0x20000000
endconst
```

The group does not create another scope. Later constants can use earlier ones. Restricted `?` inference follows the variable rules and does not remove the constant-expression requirement; a grouped `const(Type)` header cannot use `?`.

`FirstInt`, `FirstUint`, `FirstFloat`, and `FirstBool` provide optional constant fallbacks; see Compiler Directives.

2.3.5 Parameterized Expressions with `constexpr`

A `constexpr` names a single expression with typed parameters and an explicit result type. Call it with parentheses, even when it takes no arguments. Only value parameters are supported. Trailing parameters may have constant defaults. Reference parameter modes and variable argument lists are invalid.

```
constexpr EntryCount(channels : int, entriesPerChannel : int = 2) : int =
    channels * entriesPerChannel

const SCOPE_ENTRIES : int = EntryCount(4)
```

You can also write `-> ResultType` in place of `: ResultType`.

The compiler checks the body against the result type and substitutes each argument into the body when compiling a call. In a constant context, the expanded expression must evaluate to a constant. Calls can also use runtime arguments in executable code. Because a parameter may appear more than once or be omitted from the body, pass side-effecting expressions through a local variable first when you need them to run exactly once.

The body can use supported operators, casts, aggregate literals, built-ins such as `iif` and `Round`, and calls to previously declared `constexpr` expressions. Calls to ordinary functions and captures of enclosing runtime locals are rejected in the body. Pass a local value as an argument instead.

The result type supplies the context for a structure initializer, so the same expression can build table entries that carry immediate values or addresses.

```
struct SParamDef:
    flags : uint32
    var_ptr : pointer
    reserved : pointer
endstruct

const PAR_INT32_CONST : uint32 = 0x300
const PAR_INT32_RO : uint32 = 0x100
var g_counter : int32 = 0

constexpr PTENTRY_VAR(flags : uint32, v : pointer) : SParamDef = { flags, v, ? }
constexpr PTENTRY_CONST(n : int32) : SParamDef =
    PTENTRY_VAR(PAR_INT32_CONST, pointer(int(n)))

const PARAMS : [?]SParamDef = [
    PTENTRY_CONST(7),
    PTENTRY_VAR(PAR_INT32_RO, %g_counter),
```

```
    {}
]
```

Here the first entry carries the integer in the pointer field. The second entry carries the address of global storage, which the linker resolves.

Declare a `constexpr` at module scope or inside a function. Module declarations follow the usual visibility rules. Public declarations can be imported, aliased, selected with `only`, or reexported. Names in the body resolve where the expression is declared. See Modules and Packages.

2.3.6 Lexical Scope and Declaration Order

Lookup starts in the innermost active scope and proceeds outward. A declaration must be visible before a use that needs it. Forward declarations explicitly reserve function or object names when their implementations come later. Declarations in an inner block are not visible after that block finishes.

Shadowing a visible name can produce a warning. Choosing distinct names makes it easier to tell which value an expression reads. A local reference and an ordinary variable follow the same scope visibility rules.

2.3.7 Module and Object Visibility

Module declarations before `implementation` are public. Declarations after it are private unless they implement an earlier public declaration. Imported symbols participate in unqualified lookup according to the `use` modifiers.

`@.Name` explicitly selects the current module's global scope. `@module.Name` selects an imported module namespace. Object methods prioritize members and require qualification for outside value symbols unless an allowed method-local `use` makes them available. Type names remain available for type annotations and casts.

Object members have `public`, `protected`, and `private` visibility; their rules are in Objects and Properties. See Modules and Packages for import aliases, merged symbols, and reexports.

2.4. Expressions and Operators

An expression computes a typed value, such as `count + 1`, `ready and valid`, or a function call. Some expressions also identify storage that can be assigned.

```
use print

func *Main():
    var ratio : float = 3 / 2
    var quotient : int = 7 div 3
    var remainder : int = -7 rem 3
    var modulo : int = -7 mod 3
    PrintLn("{}, {}, {}, {}", [ratio, quotient, remainder, modulo])
endfunc
```

This prints `1.5, 2, -1, 2`.

2.4.1 Assignments

`=` stores a value in an assignable destination. The destination must be writable, and the value must be compatible with its type. Use `==` to compare values for equality.

```
var count : int = 3
count = 10
count += 2       // count is now 12
count =div= 3    // count is now 4
```

Modify-assignment reads the destination's current value, performs an operation, and stores the result back. The destination must therefore already be initialized and both readable and writable.

Modify-assignment statements are `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `&=`, `|=`, `=xor=`, `=div=`, `=rem=`, and `=mod=`. A word operator uses the leading `=` spelling shown here.

Assignment is a statement and does not produce a value. It cannot be used as a Boolean condition or nested to create chained assignments such as `a = b = 1`. Assignment and modify-assignment are outside the operator precedence table.

2.4.2 Arithmetic and Comparison

The tables below use *integer* for any signed or unsigned integer type, and *number* for an integer or floating-point value. `bool`, characters, enums, and pointers are separate categories; they are not implicitly treated as numbers.

2.4.2.1 Arithmetic Operators

Operator	Operands	Result	Meaning
`+`	Two numbers	Common numeric type	Addition.
`-`	Two numbers	Common numeric type; integer subtraction is signed	Subtraction.
`*`	Two numbers	Common numeric type	Multiplication.
`/`	Two numbers	Floating point	Floating-point division, including for two integers.
`div`	Two integers	Common integer type	Integer division, truncating toward zero.
`rem`	Two integers	Common integer type	Division remainder; a nonzero result has the dividend's sign.
`mod`	Two integers	Common integer type	Nonnegative modulo, including with a negative divisor.

`+` also concatenates compatible text into a `str`; see String Functions. Typed pointers support separate address arithmetic described in Pointers.

2.4.2.2 Numeric Result Types

For ordinary numeric operations, DQ first converts operands to a common type:

Operand types	Common type
Two signed integers	Signed integer at least as wide as both operands and the target's native `int`.
Two unsigned integers	Unsigned integer at least as wide as both operands and the target's native `uint`.
One signed and one unsigned integer	Signed integer at least as wide as both operands and the target's native `int`, regardless of operand order.
One integer and one floating-point value	The floating-point operand's type.
Two floating-point values	The wider floating-point type.

There are two arithmetic exceptions: integer subtraction always uses a signed result at the common calculation width, and `/` always returns floating point. Dividing two integers produces the target's default `float` type. Shifts have their own rule: the left operand is promoted to at least native width, preserving its signedness; the count does not affect the result type.

```
var small : uint8 = 10
var signed_value : int16 = -3
var sum : int16 = small + signed_value  // 7; calculation uses native int, then stores int16
var difference : int8 = small - uint8(12) // -2, although both operands are unsigned
var ratio : float = 3 / 2               // 1.5
```

A common signed type is not necessarily large enough to represent every value of an unsigned operand of the same width. Conversion retains its bits; it does not add a spare sign bit. Choose a wider type explicitly when the full unsigned range must participate in a signed calculation. Calculation widths and overflow are explained in Integer Calculations.

2.4.2.3 Division, Remainder, and Modulo

`/` performs floating-point division. `div` instead discards the fractional part toward zero, so `7 div 3` is `2` and `-7 div 3` is `-2`.

`rem` is the remainder associated with that truncating division. For valid integer division, it satisfies `a rem b == a - ((a div b) * b)`. `mod` starts with that remainder and, when it is negative, adds the magnitude of the divisor to make it nonnegative:

a	b	a div b	a rem b	a mod b
7	3	2	1	1
-7	3	-2	-1	2
7	-3	-2	1	1
-7	-3	2	-1	2

For a nonzero divisor, the modulo result is between zero, inclusive, and the divisor's magnitude, exclusive. This is useful for wrapping a possibly negative position into a positive range:

```
var position : int = -1
var slot : int = position mod 8  // 7
```

DQ uses `rem`, not `%`, for integer remainder. `%` obtains an address. `div`, `rem`, and `mod` reject floating-point operands; use an explicit rounding operation when an integer calculation is intended. Integer division by zero and signed minimum divided by `-1` have no defined result; guard these cases.

2.4.2.4 Comparison Operators

Every comparison returns `bool`.

Operator	Operands	Meaning
`==`	Compatible values	Equality.
`<>`	Compatible values	Inequality.
`<`, `<=`, `>`, `>=`	Numbers, matching character types, values of the same enum type, or compatible text values	Less than, less than or equal, greater than, greater than or equal.

Numeric comparisons use the common numeric type rules above. Two integers compare as signed if either is signed, and as unsigned if both are unsigned. If either operand is floating point, comparison is floating point.

Boolean equality requires two Boolean values. Numbers are not converted to Boolean values for comparison: `true == 1` is invalid. `!=` is not accepted. Enums compare only with the same enum type; ordering follows their ordinal values. Character comparison follows character codes, without case folding.

```
var count : int = 3
var limit : float = 3.5
var below_limit : bool = count < limit  // true: numeric conversion is allowed
var ready : bool = true
var unchanged : bool = ready == true   // true
```

Compatible pointers and object references support identity comparisons with `==` and `<>`, including comparison with `nil`; this does not compare pointed-to contents. Compatible text values support equality and ordering by unsigned byte value, with a shorter prefix coming first. Different string types can be compared; see String Comparison. General array sequence equality is not implicit. Dynamic arrays and slices can be compared with `[]` to test emptiness.

2.4.2.5 Range Comparisons

DQ supports a chain of two ordering comparisons with three operands. Both comparisons must point in the same direction:

Direction	Supported forms
Ascending	`x < y < z`, `x <= y < z`, `x < y <= z`, `x <= y <= z`
Descending	`x > y > z`, `x >= y > z`, `x > y >= z`, `x >= y >= z`

```
var temperature : int = 20
var comfortable : bool = 18 <= temperature < 25  // true
var within_limits : bool = 25 > temperature >= 18 // true
```

`lower <= value < upper` tests the same bounds as `(lower <= value) and (value < upper)`, but evaluates the middle expression only once. The final endpoint is evaluated only if the first comparison succeeds:

```
use print

var calls : int = 0

func MiddleValue() -> int:
    ++calls
    return 2
endfunc

func *Main():
    var inside : bool = 1 < MiddleValue() < 3 // calls becomes 1
    var skipped : bool = 3 < 2 < MiddleValue() // the function is not called
    PrintLn("{} {} {}", [inside, skipped, calls]) // true false 1
endfunc
```

Longer chains such as `w < x < y < z`, direction changes such as `x < y > z`, and equality chains such as `a == b == c` are invalid. Write separate comparisons joined with `and` when the condition needs those forms.

2.4.3 Integer Calculations

DQ integer expressions use fixed widths and have no overflow checks. Operands are extended to at least the target's native integer width before calculation: 32 bits on 32-bit targets and 64 bits on 64-bit targets. Wider operands retain their width, so an `int64` calculation still uses 64 bits on a 32-bit target. Signed operands are sign-extended; unsigned operands are zero-extended. A non-negative whole-number literal starts as native `uint` (or `uint64` if it exceeds 32 bits on a 32-bit target); unary negation and subtraction switch the calculation mode to signed `int` (or `int64`). See Literals and Integer Promotion for the complete rules.

```
var left : uint8 = 250
var right : uint8 = 10
var sum : int = left + right // 260: both bytes are extended before addition
var literal_sum : int = left + 10 // 260: no type hint is needed
left = left + 10             // 4: storing into uint8 keeps the low bits

var rxbuf : [2]byte = [0xFF, 0x04]
var word : uint16 = (rxbuf[0] << 8) | rxbuf[1] // 0xFF04
```

The destination does not widen a calculation retroactively. Signed narrowing interprets the retained bits as a two's-complement value. The same rules apply to compile-time and runtime expressions.

Subtraction always produces a signed result at the common calculation width, even when both operands are unsigned. Thus `uint(3) - uint(5)` is signed `-2` at the native width. An assignment to unsigned storage reinterprets the retained bits.

2.4.4 Boolean and Bitwise Operations

2.4.4.1 Logical Operators

Operator	Operands	Result	Meaning
`not value`	One `bool`	`bool`	Reverse the Boolean value.
`left and right`	Two `bool` values	`bool`	True when both operands are true.
`left or right`	Two `bool` values	`bool`	True when at least one operand is true.

`and` and `or` short-circuit. `and` evaluates its right operand only if the left is true; `or` evaluates its right operand only if the left is false. This allows a condition to guard a later operation:

```
func IsPositive(address : ^int) -> bool:
    return address <> nil and address^ > 0
endfunc
```

For `IsPositive(nil)`, the result is `false` and `address^` is not evaluated. An `or` condition can supply the opposite guard:

```
var address : ^int = nil
var missing_or_zero : bool = address == nil or address^ == 0 // true
```

There is no numeric truthiness: `not 0` and `1 and 2` are invalid. Use a comparison such as `count <> 0` to produce a Boolean value.

2.4.4.2 Bitwise Operators

Bitwise operators work on integer bits, not Boolean truth values. They require integer operands and do not implicitly round floating-point values.

Operator	Operands	Result	Meaning
`~value`	One integer	Operand's integer type promoted to at least native width	Complement every bit.
`left & right`	Two integers	Common integer type	Set bits present in both operands.
`left | right`	Two integers	Common integer type	Set bits present in either operand.
`left xor right`	Two integers	Common integer type	Set bits present in exactly one operand.
`value << count`	Two integers	Left operand's type promoted to at least native width	Shift left, filling low bits with zeroes.
`value >> count`	Two integers	Left operand's type promoted to at least native width	Shift right; signed values extend the sign bit, unsigned values fill with zeroes.

For `&`, `|`, and `xor`, signedness and width follow the common integer rules. For shifts, the count does not change the promoted left operand's type or signedness. The count must be nonnegative and smaller than the calculation width. Bits shifted out are discarded; shifts do not report overflow.

```
var a : uint8 = 0x0C       // 00001100
var b : uint8 = 0x0A       // 00001010
var both : uint8 = a & b   // 00001000 = 8
var either : uint8 = a | b // 00001110 = 14
var different : uint8 = a xor b // 00000110 = 6
var inverted : uint8 = ~a  // 11110011 = 243
var shifted : uint8 = a << 1 // 00011000 = 24; calculated at native width, then narrowed
```

To test whether any flag bits are set, compare the masked integer with zero:

```
var flags : uint = 0x12
var mask : uint = 0x02
var enabled : bool = flags & mask <> 0 // true
```

Bitwise operations bind more tightly than arithmetic and comparison. `flags & mask <> 0` therefore means `(flags & mask) <> 0`. Logical `and` and `or` bind more loosely than comparisons. Neither `&` nor `|` short-circuits.

2.4.5 Unary and Storage Operations

A unary operator acts on one operand. Most are written before it; pointer dereference is written after it:

Form	Operand	Result	Meaning
`-value`	A number	Same floating-point type, or signed integer promoted to at least native width	Numerical negation.
`not value`	`bool`	`bool`	Logical negation.
`~value`	An integer	Operand's integer type promoted to at least native width	Bitwise complement.
`%value`	Addressable storage of type `T`	`^T`	Obtain its address.
`address^`	A typed pointer `^T`	The referenced `T`	Access the value at that address.

Negating an unsigned integer produces a signed result at least as wide as native `int`. Like other fixed-width integer calculations, this can overflow when the mathematical result does not fit. Widen the operand before negation when necessary:

```
var magnitude : uint8 = 7
var negative : int8 = -magnitude // -7
var value : int = 10
var address : ^int = %value
address^ = 42                  // value becomes 42
```

The prefix `^` in `^int` is a **type designator**, not an expression operator. `^int` means pointer to `int`, and `^^int` means pointer to pointer to `int`. The postfix `^` in `address^` instead dereferences a pointer value. `%` requires an addressable operand; it cannot obtain the address of a literal or a property.

Prefix `++value` and `--value` update an assignable number or typed pointer as statements. They produce no value and do not introduce C-style postfix increment expressions.

Member access and indexing extend an expression:

Form	Operand	Result
`value.member`	A structure, object, or supported pointer	The named field, method, or property.
`array[index]`	An array and integer index	The selected element.
`pointer[index]`	A typed pointer `^T` and integer offset	Another `^T`, offset by that many elements.

Pointer indexing does not dereference. Write `pointer[index]^` to access the pointed-to element:

```
var values : [3]int = [10, 20, 30]
var first : ^int = %values[0]
var second : ^int = first[1] // points to values[1]
var value : int = second^   // 20
```

See Pointers for address arithmetic and lifetime rules, and Arrays and Slices for element access and slicing.

2.4.6 Explicit Conversion and Runtime Type Tests

`Type(expression)` and `expression as Type` request the same defined conversion. `as` binds at comparison precedence and cannot be chained with another comparison-level operator. Parenthesize its result before further operations.

`value is OType` tests an object's runtime type and returns `false` for `nil`. `TryCast(TargetType, value)` performs a checked object cast; see Objects and Properties. A cast does not validate an arbitrary pointer, encoding, or enum ordinal.

2.4.7 Inline Conditional

`iif(condition, when_true, when_false)` selects a value within an expression. It evaluates `condition` first, then evaluates and returns `when_true` if the condition is `true`, or `when_false` otherwise. The condition must have type `bool`.

```
var count : int = 0
var label : rostr = iif(count == 0, "empty", "nonempty")
```

Here `label` receives `"empty"`. The alternatives can also be numeric or other compatible value types. For example, an integer and a floating-point alternative resolve to a floating-point result:

```
var use_default : bool = false
var scale : float = iif(use_default, 1, 2.5) // 2.5; true would select 1
```

The alternatives must resolve to a common result type under DQ's conversion rules. Numeric alternatives use the numeric type rules, and a compatible typed pointer can be paired with `nil`. The result must also be assignable to its destination: a floating-point result does not implicitly become an integer merely because the destination is `int`.

Only the selected alternative runs. This makes it possible to guard an operation that would be invalid for the other case:

```
func ValueOrZero(address : ^int) -> int:
    return iif(address <> nil, address^, 0)
endfunc
```

`ValueOrZero(nil)` returns `0` without dereferencing `nil`. Calls and other side effects in the unselected alternative do not occur. An error in the selected alternative still occurs normally; `iif` does not catch exceptions.

Both alternatives are parsed and type-checked, even when the condition is a constant. An unselected alternative cannot contain an unknown name or an incompatible type. The following uses are compile errors:

Invalid expression	Reason
`iif(1, 2, 3)`	The condition is an integer, not `bool`.
`iif(true, 1, "text")`	The alternatives have no compatible common type.
`iif(true, 1)`	The false alternative is missing.
`iif(true, 1, 2, 3)`	Exactly three arguments are required.

Each alternative must produce a value; a call to a function with no result cannot serve as an alternative. Use an `if` statement when choosing between actions that do not produce values.

2.4.8 Calls and Evaluation

Calls use parentheses and pass arguments positionally, in declaration order. Arguments may themselves be expressions or calls. Trailing arguments can be omitted when the function declares defaults:

```
use print

func Subtract(left : int, right : int = 1) -> int:
    return left - right
endfunc

func *Main():
    var difference : int = Subtract(5 + 5, 3)           // 7
    var previous : int = Subtract(10)                  // 9: right defaults to 1
    var nested : int = Subtract(Subtract(10, 3), 2)    // 5
    PrintLn("{} {} {}", [difference, previous, nested])
endfunc
```

The first argument supplies `left`, and the second supplies `right`. Named arguments such as `Subtract(right = 3, left = 10)` are not supported. In the nested example, the inner call returns `7` before the outer call subtracts `2`.

Arguments are evaluated before the called function is entered. Avoid relying on a relative evaluation order between independent expressions with side effects unless a construct explicitly specifies that order. When order matters, use separate statements. With `Subtract` defined above:

```
func Next(counter : ref int) -> int:
    ++counter
    return counter
endfunc

func OrderedDifference() -> int:
    var counter : int = 0
    var first : int = Next(counter)   // counter becomes 1
    var second : int = Next(counter)  // counter becomes 2
    return Subtract(first, second)    // -1
endfunc
```

This explicitly orders the two updates instead of depending on the argument evaluation order in `Subtract(Next(counter), Next(counter))`. Reference parameters such as `ref int` are described in Functions.

Built-ins such as `Len`, `SizeOf`, `OffsetOf`, `Ord`, `Round`, `Defined`, and `FirstInt` use call-like syntax but follow their own type and constant-expression rules:

```
var values : [3]int = [10, 20, 30]
var count : int = Len(values)       // 3 elements
```

```
var bytes : uint = SizeOf(int32)    // 4 bytes
var rounded : int = Round(3 / 2)    // 2
```

`SizeOf` accepts a type name as shown here, while an ordinary value parameter expects a value expression. Compiler Directives explains compile-time existence tests and optional constants.

2.4.9 Associativity

Associativity determines grouping when operators have the same precedence. Arithmetic and bitwise binary operators group from left to right:

```
var left_grouped : int = 10 - 3 - 2   // (10 - 3) - 2 = 5
var right_grouped : int = 10 - (3 - 2) // 10 - 1 = 9
var quotient : int = 20 div 3 div 2    // (20 div 3) div 2 = 3
```

Comparisons do not form ordinary left-associative chains. The three-operand range comparison is the specific exception described in Range Comparisons. `is` and `as` likewise do not form comparison chains; parenthesize a conversion before applying another operator at that precedence level.

Grouping does not establish a general evaluation order for independent calls with side effects. Use separate statements when their order matters, as shown in Calls and Evaluation.

2.4.10 Parentheses

`(expression)` groups an expression as one operand and overrides the usual precedence. Parentheses can be nested. They do not by themselves convert its type or turn a value into addressable storage.

```
var ordinary : int = 2 + 3 * 4    // 14
var grouped : int = (2 + 3) * 4   // 20
var divided : int = 8 * 3 div 2   // 8 * (3 div 2) = 8
var multiplied : int = (8 * 3) div 2 // 12
```

The last two lines illustrate DQ's division precedence: `div` binds more tightly than `*`. Bitwise operators also bind more tightly than arithmetic:

```
var shifted_first : int = 1 << 2 + 1 // (1 << 2) + 1 = 5
var added_first : int = 1 << (2 + 1) // 1 << 3 = 8
```

Use parentheses to make the intended grouping clear when an expression mixes operator groups. This is especially helpful when translating code from a language with different precedence rules.

2.4.11 Operator Precedence

From tightest to loosest binding:

Level	Operators and forms
1	Literals, names, parenthesized expressions, casts, array literals, `new`, built-ins.
2	Calls, `.member`, indexing, slicing, postfix `^`.
3	Address-of `%`, unary `-`, bitwise `~`.
4	`<<`, `>>`
5	`&`
6	`|`, `xor`
7	`/`, `div`, `rem`, `mod`
8	`*`
9	`+`, `-`
10	`==`, `<>`, `<`, `<=`, `>`, `>=`, `is`, `as`
11	`not`
12	`and`
13	`or`

Division binds more tightly than multiplication. Bitwise operations also bind more tightly than arithmetic. Use parentheses whenever mixed operators obscure the intended grouping rather than assuming another language's precedence.

2.5. Literals and Integer Promotion

DQ applies one set of numeric rules to both runtime code generation and compile-time constant evaluation. The model is designed for predictable machine-level behavior on 32-bit and 64-bit targets—especially RISC architectures—without forcing type suffixes or redundant casts onto ordinary expressions.

```
use print

func *Main():
    var reg : uint32 = 0x80000000
    var half : uint32 = reg div 2         // 0x40000000: 2 starts as uint, so div stays unsigned
    var signed_diff : int = 3 - 5         // -2: subtraction switches calculation mode to signed int
    var rxbuf : [2]byte = [0xFF, 0x04]
    var word : uint16 = (rxbuf[0] << 8) | rxbuf[1] // 0xFF04: bytes widen to native width first
    PrintLn("0x{:X} {} 0x{:04X}", [half, signed_diff, word])
endfunc
```

2.5.1 Numeric Literals

2.5.1.1 Non-Negative Integer Literals

A non-negative decimal (`0` , `42` , `5000000000`) or hexadecimal (`0xFF` , `0x80000000` , `0x100000000`) literal without an explicit type cast starts as unsigned:

Target pointer width	Literal magnitude	Starting type
64-bit	`0` .. `0xFFFFFFFFFFFFFFFF`	`uint` (`uint64`)
32-bit	`0` .. `0xFFFFFFFF`	`uint` (`uint32`)
32-bit	`0x100000000` .. `0xFFFFFFFFFFFFFFFF`	`uint64`

Starting non-negative literals as `uint` keeps unsigned expressions unsigned when combined with plain numbers:

```
var reg : uint32 = 0x80000000
var nonzero : bool = reg > 0          // true: unsigned comparison
var half : uint32 = reg div 2         // 0x40000000: unsigned division
var shifted : uint32 = (reg | 1) >> 1 // 0x40000000: logical right shift
var cleared : uint32 = reg & ~(1 << 5) // stays native 32-bit on 32-bit targets
var all_ones : uint = ~0              // unsigned all-ones mask at native width
```

Signed expressions still work without casts because combining `uint` with a signed operand promotes to signed `int` :

```
var offset : int = -10
var moved : int = offset + 5          // -5: int + uint -> int
var positive : bool = offset > 0      // false: int > uint -> signed comparison
```

2.5.1.2 Negated Integer Literals

Applying unary `-` switches the calculation mode to signed. When `-` is applied directly to a plain non-negative literal:

Target pointer width	Negated value	Result type
64-bit	`-9223372036854775808` .. `0`	`int` (`int64`)
32-bit	`-2147483648` .. `0`	`int` (`int32`)
32-bit	`< -2147483648` (for example `-3000000000`)	`int64`

On a 32-bit target, a directly negated plain literal such as `-3000000000` or `-2147483649` promotes to `int64` so that `var x : int64 = -3000000000` keeps its exact value instead of wrapping in 32 bits. An explicitly typed negation such as `-uint32(3000000000)` respects the explicit type and negates at native 32-bit `int` .

2.5.1.3 Floating-Point Literals and Assignment Context

A literal containing a decimal point or exponent (`2.5` , `1e6` , `0.`) has the target's default `float` type (`float32` on single-precision FPU targets, `float64` otherwise).

You do not need type hints like `uint8(3)` or `2.0` when initializing or assigning variables, passing arguments, or returning values. The compiler converts the literal or expression to the destination type:

```
var limit : uint8 = 255
var scale : float = 2
var wide : uint64 = 0x100000000
var neg_wide : int64 = -3000000000
```

2.5.2 Minimum Calculation Width

DQ never performs integer calculations narrower than the target's native integer width:

- **32 bits** (`uint32` / `int32`) on 32-bit targets.
- **64 bits** (`uint64` / `int64`) on 64-bit targets.

Operands wider than the native width (`uint64` , `int64` on a 32-bit target) keep their 64-bit calculation width.

Before an integer operation runs, any narrower operand (`uint8` / `byte` , `int8` , `uint16` , `int16` , and on 64-bit targets `uint32` / `int32`) is extended to the calculation width:

- **Unsigned operands are zero-extended** (high bits filled with `0`).
- **Signed operands are sign-extended** (sign bit replicated into high bits).

```
var a : uint8 = 250
var b : uint8 = 10
var sum : int = a + b        // 260: both bytes zero-extend before addition
var high : byte = 0xFF
var low : byte = 0x04
var word : uint16 = (high << 8) | low // 0xFF04: high extends before shifting
```

The destination type of an assignment does not widen a calculation retroactively; it only narrows or reinterprets the final result when stored.

2.5.3 Integer and Floating-Point Promotion Rules

For binary operations, DQ promotes operands along the chain `uint -> int -> int64 / float` (with `uint64` preserved when both operands are unsigned and at least one is 64-bit):

Left / Right operand types	Calculation type
Both unsigned, width `<= native`	`uint` (native width)
Both unsigned, either is `uint64`	`uint64`
Both signed, width `<= native`	`int` (native width)
Both signed, either is `int64`	`int64`
One unsigned and one signed, width `<= native`	`int` (native width)
One unsigned and one signed, either is 64-bit	`int64`
One integer and one floating-point value	The floating-point operand's type
Two floating-point values	The wider floating-point type

2.5.3.1 Operator-Specific Rules

Operators	Signedness and width rule
`+`, `*`, `div`, `rem`, `mod`, `&`, `|`, `xor`	Common type from the table above. Unsigned-only stays unsigned.
`-` (binary subtraction)	Always **signed** (`int` or `int64`) at the common calculation width, even when both operands are unsigned.
`-x` (unary negation)	Always **signed** (`int` or `int64`) at `max(native, width(x))` (or `int64` for a directly negated 32-bit-overflowing plain literal).
`~x` (bitwise complement)	Promoted to `max(native, width(x))`, preserving `x`'s signedness.
`x << n`, `x >> n` (shifts)	Left operand `x` is promoted to `max(native, width(x))`, preserving `x`'s signedness (`>>` is logical for unsigned, arithmetic for signed). The count `n` does not widen `x` or change its signedness.
`/` (floating-point division)	Always floating point. Dividing two integers converts both to the target's default `float`.
`==`, `<>`, `<`, `<=`, `>`, `>=`	Operands harmonize using the table above, then compare at that common type (unsigned if both unsigned, signed if either signed, float if either float).

When an unsigned integer (including `uint64`) converts to floating point—whether via `/`, mixed integer/float arithmetic, assignment, or an explicit cast—DQ converts its **unsigned** value (`uint64(0xFFFFFFFFFFFFFFFF)` becomes `18446744073709551616.0`, not `-1.0`).

2.5.4 Overflow, Wraparound, and Storage Narrowing

DQ does not perform runtime integer range checking. Fixed-width integer arithmetic (`+`, `-`, `*`, unary `-`) uses exact two's-complement machine wraparound.

- **Same-width `int` ↔ `uint` conversion** is a pure bit-copy (zero cost).
- **Narrowing into storage** (`byte`/`uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`) truncates to the low bits. Signed storage interprets the retained bits in two's complement.

```
var x : uint8 = 250
var y : uint8 = 10
var stored_u8 : uint8 = x + y  // 260 -> low 8 bits -> 4
var stored_i8 : int8 = x + y   // 260 -> low 8 bits -> 4
var wrapped : uint8 = 0 - 1    // -1 -> low 8 bits -> 255
```

2.5.4.1 High-Bit Unsigned Subtraction and Bit-Copy Identity

Because subtraction switches the calculation mode to signed `int` (or `int64`) at the common calculation width, `u - 1` produces a signed value whose bit pattern is the exact machine subtraction result:

```
var u : uint = 0
var u2 : uint = u - 1          // all-ones (~0) via bit-copy
```

The identity below always holds for every integer type `u` (signed or unsigned, narrow or wide, 32-bit or 64-bit):

```
uint(uint(u) - uint(1)) == uint(u - 1)
```

When operating near the top of the unsigned range (bit 31 set on 32-bit, or bit 63 set on 64-bit), remember that the result of `u - 1` is signed. If you immediately follow it with a signedness-sensitive operator (`>>`, `div`, `rem`, `mod`, `<`, `<=`, `>`, `>=`), cast it back to `uint` (or `uint64`) first:

```
var addr : uint = 0x80000002
var shifted : uint = uint(addr - 1) >> 1 // logical shift on 0x80000001
```

2.5.4.2 Mixing Unsigned and Signed at Equal Width (32-Bit vs 64-Bit)

Mixing an unsigned value and a signed value at the same calculation width converts the unsigned operand to signed by bit-copy—it does **not** silently widen 32-bit operations to 64-bit on 32-bit targets.

As a consequence, `uint32(0xFFFFFFFF) > int32(-1)` is:

- `true` **on 64-bit targets**, where both narrow operands first extend to native `int64` (`4294967295 > -1`).
- `false` **on 32-bit targets**, where 32 bits is already native width and both compare as `int32` (`-1 > -1`).

When a calculation must accommodate the full `uint32` range together with negative numbers portably across 32-bit and 64-bit targets, widen explicitly to `int64`:

```
var u : uint32 = 0xFFFFFFFF
var s : int32 = -1
var portable : bool = int64(u) > int64(s) // true on both 32-bit and 64-bit
```

2.5.5 Modify-Assignment and Pointer Offsets

Modify-assignment statements (`+=`, `-=`, `*=`, `=div=`, `=rem=`, `=mod=`, `&=`, `|=`, `=xor=`, `<<=`, `>>=`) promote and calculate with the same rules as the corresponding binary expression, then narrow the result back into the destination's storage width:

```
var a : uint8 = 200
a =div= 300        // calculated at native width: 200 div 300 = 0 (not 200 div 44)

var b : uint8 = 10
b =div= -3         // calculated as signed native int: 10 div -3 = -3 -> uint8(253)

var c : uint8 = 1
c <<= 8            // calculated at native width: 1 << 8 = 256 -> uint8(0)
```

Pointer arithmetic (`p + n`, `p - n`, `p += n`, `p -= n`) and pointer indexing (`p[n]`) extend narrow integer offsets according to their signedness (zero-extending unsigned types like `uint8`/`uint16`, sign-extending signed types like `int8`/`int16`) before applying the element offset:

```
var buf : [256]byte
var base : ^byte = %buf[0]
var off : uint8 = 200
var elem : ^byte = base + off  // advances +200 bytes (zero-extended, never -56)
```

2.5.6 Compile-Time Constant Evaluation

Compile-time evaluation in `const`, `#define`, `#if`, and folded subexpressions uses the **exact same rules** as runtime code generation:

- Literals start with the same `uint` / `uint64` / `int` / `int64` types.
- Intermediate operations run at `max(native, operand_widths)` with the same signedness rules and two's-complement wraparound.
- Unsigned-to-float conversions preserve unsigned magnitude.
- Shift counts must be in `0 .. calculation_width - 1` (so `1 << 33` is valid on 64-bit targets where native width is 64, and rejected on 32-bit targets unless the left operand is `uint64(1)`).
- Untyped `#define` constants keep the expression's resulting type (`#define N = 5` is `uint`; `#define D = -5` is `int`).

```
const UMAX : uint64 = 0xFFFFFFFFFFFFFFFF
const HALF_F : float = UMAX / 2       // matches runtime UMAX / 2 exactly
const WORD : uint16 = (byte(0xFF) << 8) | byte(0x04) // 0xFF04
const MASK : uint = ~0                // matches runtime ~uint(0)
```

2.5.7 Overload Resolution and `anyvalue`

When resolving overloaded functions, DQ prefers conversions in this order:

1. **Exact type match**
2. **Integer-to-integer conversion** (for example `uint` → `int` or `uint8` → `int`)

3. **Integer-to-float conversion** (`uint` → `float`)

4. `anyvalue` **boxing**

Because integer-to-integer conversion ranks ahead of integer-to-float, calling an overloaded function `Pick(int)` / `Pick(float)` with a non-negative literal such as `Pick(5)` selects `Pick(int)`, while `Pick(2.5)` selects `Pick(float)`.

When a non-negative literal is boxed directly into `anyvalue` (for example `var v : anyvalue = 123`), it is stored as an unsigned native integer: both `v.IsInt()` and `v.IsUint()` return `true`, and `v.AsInt(0)` / `v.AsUint(0)` return `123`.

2.6. Statements and Control Flow

Statements declare storage, assign values, call functions, and choose what runs next. Conditions must be Boolean expressions. A block groups statements between `:` and its matching closer, or between braces where supported.

```
use print

func *Main():
    var total : int = 0
    for i : int = 1 to 3:
        total += i
    endfor
    if total == 6:
        PrintLn("total = {}", [total])
    endif
endfunc
```

2.6.1 Declaration and Assignment

`var`, `const`, `type`, and local `ref` declarations follow the rules in Variables, Constants, References, and Scope. Plain assignment uses `=`. Modify-assignment reads the existing value, performs an operation, and stores the result. Thus its destination must already be initialized and both readable and writable.

A property can be an assignment destination. Modify-assignment requires both a getter and setter, because the old value must be read before the update. Assignment does not produce a value.

2.6.2 Conditional Statements

```
var count : int = 3
if count == 0:
    PrintLn("empty")
elif count < 10:
    PrintLn("small")
else:
    PrintLn("large")
endif
```

Branches are considered in order; at most one body runs. `elif` and `else` are optional. Numeric values cannot serve directly as conditions.

2.6.3 While Loops

A loop repeats a group of statements, called its body. Each repetition is called an iteration. A `while` loop tests its condition before each iteration: if the condition is true, it runs the body and then tests the condition again. When the condition is false, execution continues after the loop. If the condition is false at the first test, the body does not run at all.

The body of a `while` loop is written between `while ... :` and `endwhile`:

```
var remaining : int = 3
while remaining > 0:
    PrintLn("{}", [remaining])
    remaining -= 1
endwhile
```

This loop prints `3`, `2`, and `1`, each on a separate line. Each iteration decreases `remaining` by one. Once `remaining` reaches zero, the condition `remaining > 0` is false and the loop ends.

`break` exits the nearest enclosing loop. `continue` starts its next iteration; a `while` loop then tests its condition again.

```
var number : int = 0
while number < 10:
    number += 1
    if number == 2:
        continue // Skip the rest of this iteration.
    endif
    if number == 5:
        break // Leave the loop immediately.
    endif
```

```
    PrintLn("{}", [number])
endwhile
```

This loop prints `1`, `3`, and `4`, each on a separate line. `continue` skips the print when `number` is `2`, and `break` ends the loop when it reaches `5`, even though `number < 10` is still true. The increment comes before `continue` so that `number` advances even when the rest of the body is skipped.

2.6.4 Numeric For Loops

A numeric `for` loop requires a control variable of a signed or unsigned integer type, such as `int`, `uint`, `int32`, or `uint32`. Floating-point types are not supported. A loop can declare its control variable or use an existing compatible variable. The principal forms are:

Form	Meaning
`for i : int = first to last`	Increase from `first` through inclusive `last`.
`for i : int = first downto last`	Decrease through inclusive `last`.
`for i : int = first count n`	Run `n` iterations, increasing the control value.
`for i : int = first downcount n`	Run `n` iterations, decreasing the value.
`for i : int = first while condition`	Continue while the Boolean condition is true.

Append `step amount` before the colon to change the default step of one. `to`, `downto`, `count`, and `downcount` require a positive step magnitude; the loop form supplies the direction. A `while` form can use a negative step.

```
for i : int = 0 to 6 step 2:
    PrintLn("{}", [i]) // 0, 2, 4, 6
endfor

for i : int = 5 downcount 3:
    PrintLn("{}", [i]) // 5, 4, 3
endfor

for i : int = 6 while i > 0 step -2:
    PrintLn("{}", [i]) // 6, 4, 2
endfor
```

The initial value is evaluated once on entry. `count` and `downcount` also evaluate their iteration count once. In contrast, `to` and `downto` reevaluate their limit at each test, and the `while` form reevaluates its condition. The step expression is evaluated for each step. `continue` still performs the `for` loop's normal step before the next test. Avoid changing the loop's control values from inside the body; use a `while` loop for explicit state updates.

2.6.5 For-In Loops

A `for ... in` loop goes through the elements of a fixed array, slice, or dynamic array in order, from first to last. This is called iterating over the array. On each iteration, the loop variable represents the current element and the loop body runs for that element. If the array is empty, the body does not run.

```
var values : [3]int = [10, 20, 30]
for value :? in values:
    PrintLn("{}", [value])
endfor

for value : ref in values:
    value += 1
endfor
```

The first loop prints `10`, `20`, and `30`, each on a separate line. The second loop increases each element by one, leaving `values` as `[11, 21, 31]`.

`:?` infers an ordinary element value. `: ref` binds to each element's storage, so assignment changes the array. An existing variable or an explicit element type can also be used. The source must be an addressable array value.

Do not resize a dynamic array while a reference iteration variable is in use; its element addresses can become invalid. See Arrays and Slices.

2.6.6 Return Values

`return` exits a function immediately, so statements after it are skipped. In a function with no result, use `return` on its own:

```
func PrintIfPositive(value : int):
    if value <= 0:
        return
    endif
    PrintLn("{}", [value])
endfunc
```

`PrintIfPositive(3)` prints `3`. `PrintIfPositive(0)` returns before reaching `PrintLn`, so it prints nothing.

In a value-returning function, `return expression` sends a value back to the caller. Alternatively, assign the built-in `result` variable and let execution reach `endfunc`:

```
func Add(left : int, right : int) -> int:
    return left + right
endfunc

func Sum(values : []int) -> int:
    result = 0
    for value :? in values:
        result += value
    endfor
endfunc
```

`Add(2, 3)` returns `5`, and `Sum([10, 20, 30])` returns `60`. In `Sum`, `result` holds the running total, so no separate accumulator variable is needed. It starts at zero, which is also the returned value for an empty slice. Assigning `result` does not exit the function; the loop keeps adding to it until all elements have been visited, and `endfunc` returns the total. Every normal exit of a value-returning function must have a valid result; see Functions.

2.6.7 `*Main()` Exit Status

`*Main()` has no result, so use the global `exit_code` to set the application's exit status. Its default value is `0`, which means success; a nonzero value indicates failure. Assigning `exit_code` does not stop execution. Use `return` to leave `*Main()` early:

```
use print

func *Main():
    PrintLn("Operation failed")
    exit_code = 1
    return
endfunc
```

This application exits with status `1`. You can also let execution reach `endfunc` after setting `exit_code`.

Alternatively, `Exit(n)` terminates the application immediately with status `n`, even when called from a function other than `*Main()`:

```
use print

func *Main():
    PrintLn("Operation failed")
    Exit(2)
endfunc
```

This application exits with status `2`; `Exit` does not return to its caller.

2.6.8 Exceptions and Cleanup

`try`, `except`, `finally`, and `raise` transfer control when an operation fails. `finally` runs on normal exit as well as `return`, `break`, `continue`, and exception propagation. The full rules and examples are in Exceptions and Error Handling.

2.6.9 Method-Local Use

Object methods may use `use .`, `use *`, or imported module namespaces to opt outside variables, constants, and functions into unqualified lookup. This is a restricted method feature, not a general import statement for arbitrary function bodies. See Objects and Properties.

2.7. Functions

A function gives a name and a parameter list to reusable code. Parameters carry input values; an optional result type describes the value returned to the caller.

```
use print

func Add(left : int, right : int) -> int:
    return left + right
endfunc

func *Main():
    PrintLn("{}", [Add(2, 3)])
endfunc
```

2.7.1 Declaration and Definition

A definition uses `func Name(parameters)`, an optional `-> ResultType`, and a body closed by `endfunc`. Omit `->` for a function with no result. A declaration reserves the signature without supplying its body:

```
func Add(left : int, right : int) -> int ..
```

Implement it later with the matching name, ordered parameter types and modes, result type, and relevant attributes. The public part of a module can declare functions whose definitions follow `implementation`.

For a reusable expression that can be used in a constant initializer, see `constexpr`.

2.7.2 Result Values

A function with a result type has its own built-in `result` variable. Either return a value directly or assign `result` before a normal exit:

```
func Square(value : int) -> int:
    result = value * value
endfunc
```

Every reachable normal exit must leave the result assigned. A function without a result cannot `return value`. A returned view, pointer, object reference, or bound method must not refer to storage whose lifetime ends with the function.

2.7.3 Parameter Modes

Plain parameters receive values. Copying an object reference copies its address, and copying `str` retains its value. Passing by value does not mean deep-copying an object or its data. Dynamic-array parameters require a reference mode; use `[]T` for an ordinary borrowed array parameter.

Form	Contract
`value : T`	A value parameter; rebinding it does not rebind caller storage.
`value : ref T`	A readable and writable alias to caller storage.
`value : refin T`	A read-only input alias.
`value : refout T`	An output alias whose input value is not part of the contract.
`value : refnil T`	A reference that may be omitted by passing `nil`.

```
func SetValue(value : ref int):
    value = 5
endfunc

func MaybeSet(value : refnil int):
    if %value == nil:
        return
    endif
```

```
    value = 44
endfunc
```

A reference argument needs compatible addressable storage with sufficient lifetime. A property is not itself addressable. For `refnil`, test `%value` before reading or writing when the caller may have supplied `nil`.

`[]T`, `rostr`, `strslice`, and unsized `embstr` borrow storage even when their descriptors are passed by value. A normal object/pointer parameter borrows an autofree variable; `ref` parameters preserve its `autofree` qualification exactly. Autofree type restrictions are in Memory Management and Autofree.

2.7.4 Default Arguments

Trailing parameters may have compile-time constant default values. The caller may omit those trailing arguments:

```
func Scale(value : int, factor : int = 2) -> int:
    return value * factor
endfunc

func Demonstrate() -> int:
    return Scale(4) + Scale(4, 3)
endfunc
```

This returns `20`: the calls use factors `2` and `3`. Defaults are converted to their parameter type at the declaration. Reference parameters, array-slice parameters, and unsized `embstr` parameters cannot have defaults. Calls pass arguments positionally; named arguments are not supported.

`strslice` parameters accept literal-backed constant defaults, including empty strings and named text constants:

```
func JoinItems(items : []str, separator : strslice = "|") -> str:
    return StrJoin(items, separator)
endfunc
```

2.7.5 Overloading

Functions, methods, or constructors with the same name must be explicitly marked `[[overload]]`. Parameter signatures must distinguish the alternatives:

```
func Sum(left : int, right : int) -> int [[overload]]:
    return left + right
endfunc

func Sum(left : int, right : int, third : int) -> int [[overload]]:
    return left + right + third
endfunc
```

All overloads of one name must have the same result type. Parameter names and reference modes alone do not distinguish signatures. Resolution considers argument count, available defaults, permitted conversions, and whether reference arguments meet the parameter's access requirements. An ambiguous best match is an error.

2.7.6 Variable Argument Lists

C-style variable argument lists use `...` after the fixed parameters. They let the caller pass additional arguments of different types. DQ supports this form for external functions such as C's `printf`:

```
func printf(format : ^char, ...) -> int32 [[external]] ..

func *Main():
    var answer : int32 = 42
    printf("%s: %d\n", "answer", answer)
endfunc
```

This prints `answer: 42`. The extra arguments follow the target's C calling convention, including its argument promotions. DQ cannot check that their types match the format string; see C Interoperability.

A major limitation of C-style variable arguments is forwarding: a function cannot simply pass its unnamed arguments on to another variadic function. In C, this requires a separate `va_list` interface, such as `vprintf` instead of `printf`.

A DQ function can declare a `varargs` parameter to accept a borrowed list of values of different types. The list occupies one argument position and can be forwarded as a whole.

```
use print

func Log(format : strslice, values : varargs):
    PrintLn(format, values)
endfunc

func *Main():
    Log("{}: {}", ["answer", 42])
endfunc
```

This prints `answer: 42`. The caller groups several values in brackets. A single value can be supplied directly, and `[]` supplies an empty list. `Log` forwards the list to `PrintLn`. The values remain available throughout the call, including calls made by `Log`. Use `OVarArgReader(values)` to inspect elements in a consumer. See Text Formatting for supported values.

2.7.7 Function References

A function reference lets you store a function in a variable and call it through that variable. Its type specifies the parameters and return value the function must have:

```
use print

type FTransform = func(value : int) -> int

func Double(value : int) -> int:
    return value * 2
endfunc

func Square(value : int) -> int:
    return value * value
endfunc

func *Main():
    var transform : FTransform = Double
    PrintLn("{}", [transform(4)])

    transform = Square
    PrintLn("{}", [transform(4)])
endfunc
```

This prints `8` and then `16`. The same variable first refers to `Double`, then to `Square`. Both functions take an `int` and return an `int`, so either can be assigned to `transform`.

The parameter types and modes, result, calling convention, and binding form must be compatible. A function reference can be `nil`; calling it requires a valid callable.

Explicit pointer-to-function-reference casts support low-level interfaces, but the programmer must match the real signature and ABI. They are not implicit.

2.7.8 Object Method References

A method is a function that belongs to an object. An object method reference remembers both the method and the object to call it on. Add `of object` to the function reference type:

```
use print

type FAction = func() of object

object OCounter:
    value : int = 0

    func Increment():
        value += 1
    endfunc
endobject

func *Main():
    var counter <- OCounter()
    var action : FAction = counter.Increment
    action()
    action()
    PrintLn("{}", [counter.value])
endfunc
```

This prints `2`. Assigning `counter.Increment` stores the method reference without calling it. Each later `action()` call runs `Increment` on `counter`, so it changes that object's `value`.

Keep the object alive for every call through the reference; storing its method does not extend the object's lifetime. A virtual method's implementation is resolved for that object when the reference is assigned. Pointer casts to `of object` references are not supported. See Objects and Properties.

2.7.9 Local Subfunctions

A `subfn` declares a helper inside a function or object method:

```
func Count() -> int:
    var count : int = 0
    subfn Inc(amount : int):
        count += amount
    endsubfn
    Inc(2)
    Inc(3)
    result = count
endfunc
```

`Count()` returns `5`: both calls to `Inc` add their amount to the same `count` variable in the enclosing function. Using a variable from the enclosing function is called *capturing* that variable. `Inc` works with the original `count`, so its changes are still visible in `Count` after each call.

Subfunctions follow declaration-before-use and lexical block visibility. They can access enclosing parameters, locals, constants, and the enclosing method's `self` and implicit members. Method namespace restrictions still apply: use `@.` for module-root values unless a method `use` makes them available. Parameters and locals can shadow enclosing names, with the usual warning. Each value-returning subfunction has its own `result`; an enclosing `result` is never implicitly accessible, including from a void subfunction.

The same rule applies to dynamic arrays and parameters of the enclosing function: the subfunction uses those existing variables rather than separate copies. It can also use constants from the enclosing function. Inside an object method, `self` still refers to the same object. A subfunction can call itself or another subfunction declared earlier in the same scope.

A subfunction cannot declare another subfunction, including inside control-flow blocks. Subfunctions support direct calls only: their addresses cannot be taken, and they cannot be stored, returned, or passed as function references, even when they use no variables from the enclosing function. Variables used by a subfunction keep their usual lifetime and cleanup rules. Exceptions and local cleanup follow normal function rules.

2.7.10 Special Functions

The `*` marks a function that has a special role, such as starting the application or initializing a module. It tells the compiler to check that the name is a recognized special function name.

This helps catch typos. For example, `func *ModulInit()` is rejected because `ModulInit` is not a recognized special name. Without the `*`, that misspelling could silently declare an ordinary function, which would never be called automatically to initialize the module.

Special functions must always be declared with `*` before their name. Their names are reserved: declaring them without `*` is a compiler error. Explicitly calling a special function is rarely needed, but when you do, write its name without `*`.

The special functions are:

Name	Role
`*Main()`	Application entry after runtime/module initialization; no result.
`*ModuleInit()`	Module initialization; no result.
`*Create(...)`	Object constructor.
`*Destroy()`	Object destructor.

A common use of `*ModuleInit()` is to allocate a global object on the heap so that `*Main()` can use it without any preparation:

```
use print

object OSettings:
    message : str = "Ready"
endobject

var settings : OSettings = nil

func *ModuleInit():
    settings = new OSettings()
endfunc

func *Main():
    PrintLn("{}", [settings.message])
    delete settings
endfunc
```

This prints `Ready`. Module initialization always finishes before `*Main()` runs, so `settings` already refers to the allocated object when `*Main()` starts.

The runtime function `dq_module_init()` calls the modules' `*ModuleInit()` functions once, in `use` dependency order: a used module is initialized before the module that uses it. There is no module finalization; this example releases the object explicitly with `delete settings`. See Modules and Packages.

For how to define `*Create` and `*Destroy`, including how they work with inheritance, see Constructors and Destructors.

2.8. Character and String Types

A string stores text as a sequence of bytes. A character represents one byte, one UTF-16 code unit, or one Unicode character, depending on its type. DQ provides owned strings, borrowed views, and fixed-capacity buffers so code can choose how storage is allocated and who keeps it alive. For methods, standalone helpers, and text operations, see String Functions.

Example code using strings:

```
use print

func AppendField(buffer : embstr, key : rostr, value : strslice):
    buffer.AddFmt('"{}":"{}"', [key, value])
endfunc

func *Main():
    var name : rostr = "Ada Lovelace"
    var first_name : strslice = name[0:3]
    var field : embstr(32)
    AppendField(field, "name", first_name)
    field.AddFmt(', "year":{}', [1979])

    var json : str = ""
    json.Reserve(field.length + 2)
    json.Add('{')
    json.Add(field)
    var closing : ^char = "}"
    json.Add(closing)
    PrintLn(json)
endfunc
```

The output is `{"name":"Ada", "year":1979}`. `name` borrows zero-terminated text as `rostr`, and `first_name` borrows a byte range as `strslice`. `field` owns a fixed buffer as `embstr(32)`; the unsized `embstr` parameter lets `AppendField` write into that buffer with a single `AddFmt()` call. The `Add()` accepts all string types, including `^char`.

`Reserve()` is optional: a dynamic string grows automatically as text is added. Here it reserves enough capacity for the field and both braces before assembly, improving performance by avoiding heap reallocations and copying as the string grows. It does not change the string's length or contents.

2.8.1 Character Types

Type	Meaning
`char`	unsigned 8-bit byte: one ASCII character, one character from a single-byte code page, or one UTF-8 code unit (one Unicode character uses 1...4 UTF-8 code units)
`char16`	unsigned 16-bit UTF-16 code unit (one Unicode character uses 1...2 UTF-16 code units) Recommended to use only for Windows API compatibility
`wchar`	unsigned 32-bit Unicode character value, excluding UTF-16 surrogate values

UTF-16 uses pairs of surrogate values to encode some characters. A single surrogate value does not represent a complete character on its own.

`char`, `char16`, and `wchar` are distinct types, not integer aliases. The read-only `.ord` property returns a character's integer code as `int`. Prefer `c.ord`; `Ord(c)` returns the same value. Explicit or checked conversion creates a character from an integer.

A **code unit** is one storage element of a text encoding. For example, the 😀 symbol uses four `char` bytes in UTF-8 or two `char16` values in UTF-16, but its character value fits in one `wchar`.

A character literal can initialize any character type whose range includes its value. Character variables do not implicitly convert to integers or to other character types. To convert between character types, pass through the integer code exposed by `.ord`:

```
var c8  : char   = 'A'
var c16 : char16 = '€'
var c32 : wchar  = '😀'

var code : int = c8.ord                // 65
var same_code : int = Ord(c8)          // 65: equivalent function form
```

```
var emoji_code : int = c32.ord          // 128512
var same : char = char(code)           // 'A'
var wide : wchar = wchar(c16.ord)       // '€'
var next : char = char(c8.ord + 1)      // 'B'

var checked  : char  = IntToChar(code)     // 'A'; checks the range at runtime
var fallback : wchar = IntToWchar(0xD800, '?') // '?' for an invalid character value
```

These conversions preserve the numeric code; they do not encode or decode UTF-8 or UTF-16. Explicit casts reject invalid constant values at compile time, but runtime casts can truncate to the destination width and do not validate Unicode character values. Use `IntToChar()` or `IntToWchar()` when runtime validation is needed. Without a fallback argument, an invalid value raises a runtime error.

The following examples intentionally produce compiler errors:

```
var c8     : char = 'A'
var c32    : wchar = 'A'
var code   : int = 65

c8 = code           // error: no implicit integer-to-character conversion
code = c8           // error: use c8.ord
code = int(c8)      // error: an integer cast cannot replace .ord
c32  = c8           // error: no implicit conversion between character types
c8 = char(c32)      // error: use char(c32.ord) or IntToChar(c32.ord)
c8 = c8 + 1         // error: use char(c8.ord + 1)
c8.ord = 66         // error: .ord is read-only

var too_large : char = '€'             // error: the literal does not fit in one byte
var overflow  : char = char(256)       // error: constant is outside 0 .. 255
var non_bmp   : char16 = '😀'          // error: the literal does not fit in one code unit
var surrogate : wchar = wchar(0xD800)  // error: wchar excludes UTF-16 surrogate values
```

2.8.2 Text Types

Type	Ownership and representation
`str`	Dynamic, heap-allocated, reference-counted copy-on-write byte string. May contain internal zero bytes and is not necessarily valid UTF-8. A managed type that cannot be a `struct` field. The current length is always stored.
`embstr(N)`	Embedded string: fixed-capacity, mutable, zero-terminated byte storage. A zero-byte scan may be required to calculate the length.
`embstr`	Writable, appendable borrowed reference to a shared embedded string descriptor, which carries the storage capacity. Typically used for function parameters. A zero-byte scan may be required to calculate the length.
`rostr`	Read-only borrowed, zero-terminated byte string. A zero-byte scan may be required to calculate the length.
`strslice`	Read-only borrowed view of text bytes. Zero-termination is not guaranteed. Length is always known; conversion from raw or embedded storage calculates it before creating the view.
`^char`	Raw pointer to text bytes; C string operations expect a terminating zero byte.

On targets built with dynamic strings disabled, `str` and operations that produce it are unavailable. Text literals, `rostr`, `strslice`, and `embstr(N)` remain available for text processing with borrowed or fixed storage.

2.8.2.1 The Length Calculation Problem

DQ maintains strong compatibility with C APIs, so its zero-terminated string buffers can be passed directly to C functions. C APIs that work with text usually accept a pointer to the start of a text buffer, written as `^char` in DQ. This pointer carries no information about the text's length.

Zero-terminated strings mark the end of their text with a byte whose value is zero. This is different from the printable character `'0'`. The terminating byte is not counted in the text length: `"hello"` needs five text bytes plus one zero byte.

To calculate the length from that pointer, the program must read from the start until it finds the zero byte. Longer text takes more work, and repeated length calculations can scan the same bytes again.

Storing the byte count avoids this work. DQ's string types handle this in different ways, using stored or cached lengths where possible, as described below.

2.8.2.2 `str`: Dynamic Strings

`str` allocates heap memory for its contents. As the contents grow, the backing memory block may be reallocated. The current length is always known. A trailing zero byte maintains C string compatibility. The contents can also hold arbitrary binary data, including internal zero bytes. It is a **managed type**: the runtime tracks references and frees the memory when the string data is no longer used. A `str` cannot be a `struct` field. Its maximum text length is `2147483647` bytes. Its `.length` reads the stored byte count without scanning the text.

An empty `str` may require no heap allocation. Internally, it can be represented by a `nil` pointer, indicating that no dynamic string handler object has been allocated for it yet.

Assigning a `str` shares storage. Mutation detaches when necessary so other string values retain their contents.

```
var a : str = "abc"
var b : str = a
b[0] = 'X'       // a remains "abc"

var c : str = a.Clone() // independent storage immediately
c[0] = 'Y'              // a remains "abc"; b remains "Xbc"

c.Reserve(32)           // space for at least 32 text bytes; c is still "Ybc"
c.Add("def")           // reuses the reserved storage; c is now "Ybcdef"
```

2.8.2.3 `embstr(N)` and `embstr`: Embedded Strings

`embstr(N)` provides fixed storage of `N` bytes. It holds up to `N - 1` text bytes plus a terminating zero byte, making it compatible with C strings. The position of the terminating zero determines the text length, so calculating the length may require a scan at runtime. DQ can cache the length to avoid repeated scans. Embedded strings can be stored in the program's data segment, on the stack, or inside structures and objects. Their maximum text length is `16777215` bytes.

A declaration without an initializer creates an empty string. `.Add()` and `.AddFmt()` can be called immediately after the declaration:

```
var s : embstr(16)
s.Add("value=")           // = "value=": contents of s
s.AddFmt("{}", [42])      // = "value=42": contents of s
```

For a declaration without an initializer, only the leading zero terminator is guaranteed; bytes beyond it are unspecified.

For `embstr(N)`, `.storage_size` is `N` and `.maxlength` is `N - 1`. Assignment and append operations truncate to fit the fixed capacity:

```
var fixbuf : embstr(8) = "abc"
fixbuf.Append("defghi") // stores "abcdefg"
```

Use an unsized `embstr` parameter to modify an existing fixed buffer through its shared descriptor. It remains writable and appendable: character writes, assignment, `Add()`, and `AddFmt()` modify the original buffer within its fixed capacity.

2.8.2.4 `rostr`: Read-Only Zero-Terminated Strings

Use `rostr` for read-only string parameters when zero-termination is required. It borrows text literals, `str`, `embstr(N)`, unsized `embstr`, or `^char` without copying the text bytes or retaining their storage. Its descriptor can be reassigned, but character writes and writable character references through it are rejected.

A `rostr` carries a pointer and either a known length or an indication that the length is unknown. Borrowing a literal or a `str` preserves its known length. Borrowing a raw `^char` does not scan immediately; the first operation needing the length, such as `.length`, scans and caches it in that `rostr` descriptor.

2.8.2.5 `strslice`: String Slices

Use `strslice` for read-only byte ranges with explicit bounds. It borrows text without copying its bytes or retaining its storage. A slice can describe an unterminated range or a range containing zero bytes.

Its length is always known, so `.length` reads the stored byte count directly. Conversion from raw or embedded storage resolves an unknown length before creating the slice; embedded-string scans stop at the buffer's capacity. Converting a `rostr` with an unknown length also resolves that length first.

Use `StrSlice(data, length)` to supply the length explicitly and avoid scanning:

```
var data : [3]char = ['A', char(0), 'B']
var view : strslice = StrSlice(%data[0], data.length)
```

2.8.2.6 `^char`: Raw C String Pointers

Use `^char` for raw C interfaces. It carries only a pointer, with no length or ownership information. Converting it to a borrowed string cannot recover a known length from the pointer alone. If the length is already known, use `StrSlice(data, length)` to pass it explicitly.

`.pchar` exposes a borrowed `^char` for compatible zero-terminated text storage.

```
func puts(text : ^char) -> int32 [[external]] ..

func *Main():
    var owned : str = "dynamic text"
    var readonly : rostr = "read-only text"
    var buffer : embstr(16) = "fixed text"

    var owned_ptr : ^char = owned.pchar
    var readonly_ptr : ^char = readonly.pchar
    var buffer_ptr : ^char = buffer.pchar
    puts(owned_ptr)
    puts(readonly_ptr)
    puts(buffer_ptr)
endfunc
```

This prints `dynamic text`, `read-only text`, and `fixed text` on separate lines. Each pointer refers to the source's existing text bytes; no text is copied. The `.pchar` property is also available on unsized `embstr` references.

If the string's text moves to a different memory location, for example when a dynamic string grows, the old pointer must no longer be used. Get a new pointer with `.pchar` after making the change. A nil `^char` is not valid where a C string is required unless the called API explicitly defines nil handling. See C Interoperability for passing text to C functions.

2.8.2.7 Text Descriptors

The compiler carries text pointers together with metadata in small structures called **descriptors**. For calls with `embstr`, `rostr`, or `strslice` arguments, it creates temporary descriptors as needed during argument conversion. The compiler may pass either a pointer to the descriptor or its members directly as function arguments.

These descriptors are invisible to the user, the compiler creates, updates, and passes them automatically.

For `embstr`, the runtime uses the following descriptor structure:

```
struct SDqTextInfo:
    dataptr : ^char   // points to the first text byte
    charlen : int32   // known byte length, or a negative value when unknown
    info    : uint32  // bits 0..23: maximum length; bits 24..31: flags
endstruct
```

For `rostr` and `strslice`, the runtime uses a more compact descriptor:

```
struct SDqRoStrInfo:
    dataptr : ^char
    charlen : int32   // rostr: negative means unknown; strslice: always non-negative
endstruct
```

`charlen` counts text bytes, excluding any terminating zero byte. A non-negative value indicates a known length; `0` represents a known empty string. For `embstr` and `rostr`, a negative value means that the length must be calculated when needed. For `strslice`, the length is always known and non-negative.

2.8.2.8 Conversions and Borrowed Lifetimes

Destination	Relevant rule
`str`	Owns/retains its text value; accepts compatible text views.
`rostr`	Requires existing terminated storage (like a string literal), borrows it.
`strslice`	Requires existing storage, borrows a byte range with an always-known length; conversion resolves an unknown source length first. No terminator is promised.
`embstr(N)`	Creates an embedded fixed-storage of N bytes. Written texts are truncated to fit.
Unsized `embstr`	Shares a fixed-storage, writeable embedded string.
`^char`	A raw C pointer to an existing storage with no automatic lifetime extension.

Slicing a `rostr` by byte offsets returns `strslice`, including full-range and suffix slices. A `rostr` converts to `strslice`, resolving any unknown length first. There is no direct conversion from `strslice` to `rostr`, even by a cast. When terminated storage is needed for a slice, first copy it into an owning `str` or a fixed-capacity `embstr(N)`.

Both `rostr` and `strslice` convert to owned `str` or fixed-capacity `embstr(N)`. These conversions copy the byte range, truncating only when the fixed buffer reached its capacity.

```
var owned    : str = "hello"
var borrowed : rostr = owned
var part     : strslice = borrowed[1:4]  // borrows "ell"
var copied   : str = part             // owns a copy of "ell"
```

Borrowed text must not outlive its source. The owner must stay alive, and its storage and contents must remain stable while a read-only view is used. An unsized `embstr` requires live, writable source storage. Neither a cast nor storing a view in `anyvalue` extends the owner's lifetime. Returning a view into a local buffer is invalid.

See Memory Management and Autofree for managed values and borrowed lifetimes.

2.8.3 Literals

Double quotes always represent text. A single-quoted literal containing exactly one Unicode character is `wchar`; empty or longer single-quoted literals are text.

```
var path       : str = "/"
var slash      : wchar = '/'
var byte_slash : char = '/'
var greeting   : rostr = "Hello, world!"
var field      : rostr = '"name":"Amy"'
```

The `byte_slash` initialization is accepted because the literal's character code is known at compile time and is below 256. In general, conversion from `wchar` to `char` is explicit. Single quotes around longer text let it contain double quotes without escaping them, as in `field`.

2.8.3.1 Multiline Text

For text that spans several lines, put three double quotes (`"""`) or three single quotes (`'''`) around it:

```
var message : rostr = """
    First line.
    Second line.
    """
PrintLn(message)
```

This prints:

```
First line.
Second line.
```

The text starts with "F" and ends with the final ".". Putting the quotes on separate lines does not add an empty line before or after the text. The line break between `First line.` and `Second line.` is part of the string, just as if you had written `"First line.\nSecond line."`.

You can indent the text to keep the source code readable. DQ uses the spaces before the closing quotes to decide how much indentation to remove. Extra spaces are kept:

```
var indented : rostr = '''
    First line.
      Indented second line.
    '''
PrintLn(indented)
```

There are four spaces before the closing quotes, so DQ removes four spaces from the start of each text line. The second line has six spaces, so two remain:

```
First line.
  Indented second line.
```

This also makes longer text, such as JSON, easier to write. Ordinary double quotes inside triple double quotes do not need a backslash:

```
var document : rostr = """
    {
      "name": "Amy",
      "year": 1999
    }
    """
PrintLn(document)
```

This prints:

```
{
  "name": "Amy",
  "year": 1999
}
```

Escape sequences still work inside multiline text: `\n` adds a line break, `\t` adds a tab, and `\\` adds one backslash. Three matching quotes end the text; the indentation and line-break rules are the same for both quote styles.

2.9. String Functions

Use the reference tables to find string properties, methods, helpers, and operators, then follow their links to detailed explanations and examples. For storage, conversions, and lifetime rules, see Character and String Types.

2.9.1 Properties and Methods Listing

In the **Types** column, `*` means all string types: `str`, `rostr`, `strslice` and `embstr`.

Properties are read-only. A result of `—` means that the method modifies the string and returns no value. Lengths, positions, and counts are in **bytes**, except for the explicitly Unicode-oriented members `.wclen` and `.wchar[...]`.

Property or method	Types	Result	Purpose
`.Add(value)`	`str`, `embstr`	—	Append text or one `char` byte
`.AddFmt(fmt, args)`	`str`, `embstr`	—	Append formatted values
`.Append(value)`	`str`, `embstr`	—	Alias for `.Add(value)`
`.AppendChar(ch)`	`str`, `embstr`	—	Append one `char` byte
`.capacity`	`str`	`int`	Allocated capacity for text bytes
`.Clear()`	`str`, `embstr`	—	Remove all contents; retain storage
`.Clear(free_storage)`	`str`	—	Optionally release the string's storage
`.Clone()`	`str`	`str`	Create an independent copy immediately
`.Compact()`	`str`	—	Reduce capacity to the current length
`.Contains(needle)`	`*`	`bool`	Test whether text contains a value
`.Delete(index, count = 1)`	`str`, `embstr`	—	Remove bytes
`.EndsWith(suffix)`	`*`	`bool`	Test the end of the text
`.IndexOf(needle, start = 0)`	`*`	`int`	First matching byte offset, or `-1`
`.Insert(index, value)`	`str`, `embstr`	—	Insert text or a byte at a byte position
`.LastIndexOf(needle)`	`*`	`int`	Last matching byte offset, or `-1`
`.length`	`*`	`int`	Current byte length, excluding the terminator
`.Lower()`	`*`	`str`	Copy with ASCII letters in lowercase
`.LPad(length, fill)`	`*`	`str`	Copy with padding on the left
`.LTrim()` `.LTrim(set)`	`*`	`str`	Copy without matching bytes at the left end
`.maxlength`	`embstr`	`int`	Maximum number of text bytes that fit
`.pchar`	`*`	`^char`	Borrow a pointer to the text bytes
`.Pop()`	`str`	`char`	Remove and return the last byte
`.Pop(count)`	`str`	`str`	Remove and return bytes from the end
`.PopFirst()`	`str`	`char`	Remove and return the first byte
`.PopFirst(count)`	`str`	`str`	Remove and return bytes from the beginning
`.Prepend(value)`	`str`, `embstr`	—	Insert text or a byte at the beginning
`.refcount`	`str`	`int`	Number of references sharing the string handler
`.Reserve(capacity)`	`str`	—	Ensure room for at least this many text bytes
`.RPad(length, fill)`	`*`	`str`	Copy with padding on the right
`.RTrim()` `.RTrim(set)`	`*`	`str`	Copy without matching bytes at the right end
`.Set(value)`	`str`, `embstr`	—	Replace the contents
`.SetCapacity(capacity)`	`str`	—	Set capacity, shortening the text if necessary
`.SetLength(length, fill)`	`str`	—	Resize, filling new bytes with a `char`

Property or method	Types	Result	Purpose
`.StartsWith(prefix)`	`*`	`bool`	Test the beginning of the text
`.storage_size`	`embstr`	`int`	Buffer size including space for the terminator
`.ToFloat()` `.ToFloat(defvalue)`	`*`	`float`	Parse a number; raise on invalid input or return a fallback
`.ToInt()` `.ToInt(defvalue)`	`*`	`int`	Parse an integer; raise on invalid input or return a fallback
`.ToWchars()`	`*`	`[*]wchar`	Decode the whole UTF-8 string into Unicode characters
`.Trim()` `.Trim(set)`	`*`	`str`	Copy without matching bytes at either end
`.Truncate(length)`	`str`	—	Shorten without growing
`.Upper()`	`*`	`str`	Copy with ASCII letters in uppercase
`.wchar[index]`	`*`	`wchar`	Read one Unicode character
`.wchar[start:end]`	`*`	`[*]wchar`	Decode a range of Unicode characters
`.wclen`	`*`	`int`	Count Unicode characters in UTF-8 text

Methods returning `str` require dynamic-string support even when called on an embedded string or borrowed view. Numeric conversion, search methods, embedded-string mutations, and Unicode decoding remain available with dynamic strings disabled. Using `str` always requires dynamic-string support.

For equivalents of `substring()`, `substr()`, `slice()`, `Left()`, `Right()`, or `Mid()` from other languages, see Byte Indexing and Slicing. Use `text[start:end]` to extract a substring, or `text[start:start + count]` when you have a starting position and a byte count. Positions are zero-based byte offsets, and `end` is excluded. For Unicode character positions, see Unicode Indexing and Slicing.

2.9.2 Standalone String Helpers Listing

These functions are available through the implicit `sys` runtime interface; ordinary modules need no `use` declaration. Text parameters accept the string types through their normal conversions. Functions returning `str` require dynamic-string support.

Function	Result	Purpose
`Format(fmt : strslice, args : varargs)`	`str`	Create formatted text
`PcharLen(pc : ^char)` `PcharLen(pc : ^char, maxlen : uint)`	`int`	Count bytes in a raw C string, optionally up to a limit
`StrFromUtf16(chars : []char16)`	`str`	Convert a UTF-16 array to UTF-8
`StrFromUtf16Fallback(chars : []char16, fallback_wchar : wchar)`	`str`	Convert UTF-16, replacing malformed sequences
`StrFromUtf16Ptr(chars : ^char16, count : int)`	`str`	Convert an explicit number of UTF-16 code units
`StrFromUtf16PtrFallback(chars : ^char16, count : int, fallback_wchar : wchar)`	`str`	Convert a UTF-16 pointer range with replacement
`StrFromWchars(chars : []wchar)`	`str`	Encode Unicode characters as UTF-8
`StrJoin(astr : []str, ajoiner : strslice)`	`str`	Join strings with a separator between elements
`StrLower(text : strslice)`	`str`	Copy with ASCII letters in lowercase
`StrReplace(value : strslice, old_text : strslice, new_text : strslice)`	`str`	Replace all non-overlapping occurrences of literal text
`StrSlice(data : ^char, length : int)`	`strslice`	Borrow a pointer and an explicit byte count
`StrSliceCmp(a : strslice, b : strslice)`	`int`	Compare two byte ranges
`StrSplit(astr : strslice, asplitter : strslice)`	`[*]str`	Split text at each occurrence of a separator
`StrToFloat(text : strslice)` `StrToFloat(text : strslice, defvalue : float)`	`float`	Parse a number; raise on invalid input or return a fallback
`StrToInt(text : strslice)` `StrToInt(text : strslice, defvalue : int)`	`int`	Parse an integer; raise on invalid input or return a fallback
`StrUpper(text : strslice)`	`str`	Copy with ASCII letters in uppercase

2.9.3 Text Operators

`+` concatenates compatible text operands into a new `str`. `+=` appends to a dynamic string. Both require dynamic-string support.

```
var greeting : str = "Hello, " + "world"
greeting += "!"          // "Hello, world!"
```

Using `+` repeatedly to assemble text creates temporary strings and can cause repeated heap allocations and copies. Prefer `.Add()` for plain text and `.AddFmt()` for formatted values, with `.Reserve()` when the expected size can be estimated.

2.9.4 String Comparison

Text values support `==`, `<>`, `<`, `<=`, `>`, and `>=` across different string types, including embedded strings with or without an explicit capacity. Strings are compared byte by byte, using unsigned byte values. The first difference determines the order; when one string is a prefix of the other, the shorter string comes first.

```
var name : str = "alpha"
var limit : rostr = "beta"
var before : bool = name < limit       // true
var prefix : bool = "app" < "apple"    // true
var inside : bool = "a" <= name < "z" // true
var equal : bool = name == "alpha"     // true
```

Comparison is case-sensitive. It does not validate UTF-8, normalize Unicode, or apply language-specific sorting rules. Known string lengths include internal zero bytes; raw C strings end at their first zero.

2.9.4.1 StrSliceCmp

```
StrSliceCmp(a : strslice, b : strslice) -> int
```

`StrSliceCmp(a, b)` provides the same bytewise comparison as a function. It returns a negative value if `a` comes before `b`, zero if equal, and a positive value if `a` comes after `b`. It compares the entire known byte range without requiring a terminator.

```
var order : int = StrSliceCmp("app", "apple") // negative
var same : int = StrSliceCmp("abc", "abc")   // 0
var a : ^char = "alpha"
var b : ^char = "beta"
var before : bool = StrSliceCmp(a, b) < 0     // true: compares text
```

A raw `^char` operand compares text when paired with a string value or, for ordering, a text literal. Two raw pointer variables compare their addresses; use `StrSliceCmp` to compare their text contents explicitly.

2.9.5 Standalone String Helpers

2.9.5.1 Format

```
Format(fmt : strslice, args : varargs) -> str
```

`Format(fmt, args)` creates a new string using the same placeholders as `.AddFmt()`. Its `strslice` format parameter accepts substring views directly, including text without a terminating zero. See Text Formatting for the full syntax.

```
var label : str = Format("value={:04X}", [255]) // "value=00FF"
```

2.9.5.2 StrToInt and StrToFloat

```
StrToInt(text : strslice) -> int
StrToInt(text : strslice, defvalue : int) -> int
StrToFloat(text : strslice) -> float
StrToFloat(text : strslice, defvalue : float) -> float
```

`StrToInt` parses decimal digits with an optional leading `+` or `-` and rejects values outside the range of `int`. `StrToFloat` accepts an optional leading sign, a dot or comma decimal separator, and an optional `e`/`E` exponent. An exponent may have a leading sign and must contain digits. The entire input must match; surrounding whitespace and internal zero bytes are rejected. Use `.Trim()` first when surrounding whitespace should be ignored.

Without `defvalue`, invalid or empty input raises `EStrError`. Supplying `defvalue` returns that fallback on conversion errors. When exceptions are disabled, the no-fallback form reports a runtime error and returns zero if the error handler returns; the fallback form still returns `defvalue`.

```
var count : int = StrToInt("+42")              // 42
var fallback : int = StrToInt("invalid", -1)   // -1
var value : float = StrToFloat("1.25e2")       // 125
var comma : float = StrToFloat("1,5")          // 1.5
var invalid : float = StrToFloat("1e+", -1)    // -1: missing exponent digits
```

2.9.5.3 StrUpper and StrLower

```
StrUpper(text : strslice) -> str
StrLower(text : strslice) -> str
```

These functions copy the text, converting only ASCII letters. They leave non-ASCII bytes unchanged and do not modify the source. The methods `.Upper()` and `.Lower()` provide the same conversions.

```
var text : rostr = "Hello, é!"
var upper : str = StrUpper(text) // "HELLO, é!"
var lower : str = StrLower(text) // "hello, é!"
```

2.9.5.4 StrSplit and StrJoin

```
StrSplit(astr : strslice, asplitter : strslice) -> [*]str
StrJoin(astr : []str, ajoiner : strslice) -> str
```

`StrSplit` matches the whole separator from left to right, without overlapping matches, and returns owned strings. It preserves empty fields, including at the beginning and end. Empty input produces one empty field. An empty separator raises `EStrError`; with exceptions disabled it reports a runtime error and returns an empty array if the error handler returns.

`StrJoin` inserts the joiner between array elements, including empty elements. An empty array produces an empty string; an empty joiner concatenates the elements directly. Both functions require dynamic-string support and handle explicit byte ranges, including internal zero bytes.

```
var parts : [*]str = StrSplit("one::two::::", "::") // ["one", "two", "", ""]
var joined : str = StrJoin(parts, "/")             // "one/two//"
```

2.9.5.5 StrReplace

```
StrReplace(value : strslice, old_text : strslice, new_text : strslice) -> str
```

Returns a string with every non-overlapping occurrence of `old_text` replaced by `new_text`, scanning the original input from left to right. Matching is literal and case-sensitive; replacement text is not searched again. An empty `new_text` removes matches. The input is unchanged.

An empty `old_text` raises `EStrError`, consistent with `StrSplit`'s empty separator handling. With exceptions disabled it reports a runtime error and returns an empty string if the error handler returns. The function requires dynamic-string support and accepts explicit byte ranges, including internal zero bytes.

```
var replaced : str = StrReplace("one::two::", "::", "/") // "one/two/"
var removed : str = StrReplace("a--b--", "--", "")       // "ab"
```

2.9.5.6 PcharLen

```
PcharLen(pc : ^char) -> int
PcharLen(pc : ^char, maxlen : uint) -> int
```

`PcharLen(pc)` counts bytes up to the first zero byte, excluding that byte. The bounded overload reads at most `maxlen` bytes and returns `maxlen` if no zero byte occurs in that range. Both overloads return zero for `nil`, and a zero limit reads no bytes. The byte count must fit in `int`.

2.9.5.7 StrSlice

```
StrSlice(data : ^char, length : int) -> strslice
```

`StrSlice(data, length)` borrows exactly `length` bytes without scanning for a terminator or copying the text. The memory must remain alive and stable. A negative length raises a runtime error; a `nil` pointer is valid only with a zero length.

```
var data : [3]char = ['A', char(0), 'B']
var view : strslice = StrSlice(%data[0], data.length)
var bytes : int = view.length // 3: the internal zero does not end the view
```

2.9.5.8 StrFromWchars

```
StrFromWchars(chars : []wchar) -> str
```

`StrFromWchars(chars)` encodes Unicode characters as UTF-8. Invalid Unicode character values, such as surrogate values, raise a runtime encoding error.

```
var chars : [3]wchar = ['A', 'é', '€']
var text : str = StrFromWchars(chars[:]) // "Aé€"
```

2.9.5.9 StrFromUtf16 and StrFromUtf16Ptr

```
StrFromUtf16(chars : []char16) -> str
StrFromUtf16Ptr(chars : ^char16, count : int) -> str
```

`StrFromUtf16(chars)` converts a UTF-16 array slice to UTF-8, ignoring one final zero code unit if present. Internal zeros are preserved. `StrFromUtf16Ptr(chars, count)` converts exactly `count` code units, including any zeros in that range. Surrogate pairs represent one Unicode character; malformed UTF-16 raises a runtime encoding error. The pointer form returns empty text for `nil` or a non-positive count.

```
var units : [4]char16 = [char16(65), char16(0), char16(66), char16(0)]
var text : str = StrFromUtf16(units[:])          // "A\0B": 3 bytes
var prefix : str = StrFromUtf16Ptr(%units[0], 1) // "A"
```

2.9.5.10 UTF16 Fallback Conversion

```
StrFromUtf16Fallback(chars : []char16, fallback_wchar : wchar) -> str
StrFromUtf16PtrFallback(chars : ^char16, count : int, fallback_wchar : wchar) -> str
```

`StrFromUtf16Fallback` and `StrFromUtf16PtrFallback` use the same input bounds as their strict counterparts but replace malformed UTF-16 sequences with `fallback_wchar`. Passing `wchar(0)` as the fallback skips malformed sequences.

```
var units : [3]char16 = [char16(0xD800), char16(65), char16(0)]
var replaced : str = StrFromUtf16Fallback(units[:], '?') // "?A"
var skipped : str = StrFromUtf16PtrFallback(%units[0], 2, wchar(0)) // "A"
```

2.9.6 String Properties

2.9.6.1 Length

`.length` counts text bytes, including internal zero bytes when the source has a known length. It excludes the trailing terminator. `str` and `strslice` store their lengths; embedded strings and `rostr` may need to scan for the first zero byte when the length is unknown.

```
var text : str = "Aé€"
var bytes : int = text.length  // 6 bytes
var empty : str
var zero : int = empty.length // 0, even without an allocated handler
```

2.9.6.2 Capacity

`.capacity` is the number of text bytes a `str` can hold before it needs more storage. It excludes the extra terminating zero byte and can exceed `.length`. An empty string with no allocated handler has capacity zero.

```
var text : str = "abc"
text.Reserve(100)
var length : int = text.length     // 3
var has_room : bool = text.capacity >= 100 // true
```

2.9.6.3 Embedded Storage Size

For `embstr(N)`, `.storage_size` is `N` and `.maxlength` is `N - 1`. An unsized `embstr` reports these values for the buffer it references.

```
var buffer : embstr(8) = "abc"
var size : int = buffer.storage_size // 8
var limit : int = buffer.maxlength   // 7
buffer.Add("defghi")                 // buffer is "abcdefg": truncated to fit
```

2.9.6.4 Reference Count

`.refcount` reports how many references share a dynamic string handler. Assigning one `str` to another shares the handler; modifying one value detaches its storage when necessary. A string with no handler has reference count zero. Temporary values and function arguments can also retain references.

```
var a : str = "abc"
var b : str = a
var shared : int = a.refcount // 2
b[0] = 'X'                    // a is "abc"; b is "Xbc"
var unique : int = a.refcount // 1
```

2.9.6.5 Raw Byte Pointer

`.pchar` borrows a pointer to the first text byte without copying or retaining storage. `str`, `rostr`, and embedded strings provide terminated storage; a `strslice` pointer is **not guaranteed to be zero-terminated**. Copy a slice to `str` or `embstr(N)` before passing it to a C API that expects a C string.

```
var text : str = "hello"
var pointer : ^char = text.pchar
var first : char = pointer^ // 'h'

var part : strslice = text[:2] // "he", followed by more source bytes
var terminated : str = part
var c_text : ^char = terminated.pchar // points to "he" plus a zero terminator
```

Keep the source alive and its storage stable while using the pointer. Obtain a new pointer after a mutation that can move the text. For `str`, `.pchar` returns terminated empty text even when the internal handler pointer is `nil`. C string functions see only the prefix before the first zero byte. See C Interoperability.

2.9.7 String Methods

These methods modify `str` or embedded string storage. Dynamic strings grow as needed and preserve other values that share their storage. Embedded strings keep their fixed capacity and truncate inserted or appended text to fit. An unsized `embstr` modifies the original buffer through its shared descriptor.

Text-source arguments accept `str`, `rostr`, `strslice`, `embstr(N)`, unsized `embstr`, or `^char`. `.Set`, `.Add`, `.Append`, `.Prepend`, and `.Insert` also accept one `char` byte. A `wchar` variable must be encoded as UTF-8 to append its text; it is not a single-byte source.

2.9.7.1 .Set

```
.Set(value : strslice)
.Set(value : char)
```

`.Set(value)` replaces the contents. For text values, assignment provides the same replacement syntax. Embedded destinations truncate the source to fit.

```
var text : str = "old"
text.Set("new")           // "new"
text.Set('!')             // "!"
var buffer : embstr(8)
buffer.Set("abcdefghij")  // "abcdefg"
```

2.9.7.2 .Add, .Append, and .AppendChar

```
.Add(value : strslice)
.Add(value : char)
.Append(value : strslice)
.Append(value : char)
.AppendChar(ch : char)
```

`.Add(value)` and `.Append(value)` append text or a `char` byte. `.AppendChar(ch)` accepts only a `char`. These methods modify the string and return no value. An `embstr(N)` declaration starts empty, so it can be appended to immediately.

```
var text : str = "abc"
text.Add("def")           // "abcdef"
text.Append('!')          // "abcdef!"
text.AppendChar('?')      // "abcdef!?"
```

2.9.7.3 .AddFmt

```
.AddFmt(fmt : strslice, args : varargs)
```

`.AddFmt(fmt, args)` appends text containing formatted values. The format parameter is `strslice`, so substring views can be passed directly. An `embstr(N)` declaration starts empty and can be used directly with `.AddFmt()`. Embedded destinations truncate the output to fit. Placeholder syntax and supported value types are explained in Text Formatting.

```
var text : str = "value="
text.AddFmt("{:04X}", [255]) // "value=00FF"
var buffer : embstr(16)
buffer.AddFmt("{} items", [3]) // "3 items"
```

2.9.7.4 .Prepend and .Insert

```
.Prepend(value : strslice)
.Prepend(value : char)
.Insert(index : int, value : strslice)
.Insert(index : int, value : char)
```

`.Prepend(value)` inserts at the beginning. `.Insert(index, value)` inserts before the specified byte position. The insertion position is clamped to `0 .. text.length`; `$end` inserts at the end.

```
var text : str = "bc"
text.Prepend('a')         // "abc"
text.Insert(1, "--")      // "a--bc"
text.Insert($end, '!')    // "a--bc!"
text.Insert(100, '?')     // "a--bc!?": position clamped to the end
```

Performance: `.Insert()` shifts the bytes after the insertion position to make room. Inserting near the beginning of a large string can take significant time. `.Prepend()` moves all existing text bytes.

2.9.7.5 .Delete

```
.Delete(index : int, count : int = 1)
```

`.Delete(index, count = 1)` removes up to `count` bytes starting at `index`. The position is clamped to `0 .. text.length`, and the count is limited to the remaining bytes. A non-positive count or a position at the end removes nothing.

```
var text : str = "abcdef"
text.Delete(1)            // "acdef"
text.Delete(2, 100)       // "ac": removes the entire remaining suffix
text.Delete(100)          // "ac": unchanged
```

Performance: `.Delete()` shifts the bytes after the removed range to close the gap. Deleting near the beginning of a large string can take significant time because most of the remaining text must move.

2.9.7.6 .Clear

```
.Clear(free_storage : bool = false)  // str
.Clear()                             // embstr
```

`.Clear()` makes the string empty and retains its storage for reuse. For `str`, this is equivalent to `.Clear(false)`. `.Clear(true)` releases that value's reference to the dynamic storage and leaves its length and capacity at zero. Other strings sharing the old storage retain their text. Embedded strings support only the no-argument form.

```
var text : str = "abc"
text.Reserve(100)
text.Clear()              // empty; capacity is retained
text.Add("reused")
text.Clear(true)          // empty; capacity is 0
var buffer : embstr(16) = "abc"
buffer.Clear()            // empty; the 16-byte buffer remains
```

2.9.7.7 .SetLength and .Truncate

```
.SetLength(length : uint, fill : char)
.Truncate(length : uint)
```

`.SetLength(length, fill)` sets the byte length, using the supplied `char` for new bytes when growing. Shortening discards the suffix. `.Truncate(length)` only shortens: a length greater than the current length leaves the text unchanged. Lengths must be non-negative.

```
var text : str = "abc"
text.SetLength(5, '.')    // "abc.."
text.SetLength(4, ' ')    // "abc."
text.Truncate(2)          // "ab"
text.Truncate(100)        // "ab": does not grow
```

2.9.7.8 .Pop and .PopFirst

```
.Pop() -> char
.Pop(count : int) -> str
.PopFirst() -> char
.PopFirst(count : int) -> str
```

`.Pop()` removes and returns the last `char`; `.PopFirst()` removes and returns the first. These no-argument forms raise a runtime error on an empty string.

With a count argument, both methods return a `str` containing the removed bytes. Counts greater than the length remove everything. A non-positive count or an empty string returns empty text.

```
var text : str = "abcdef"
var tail : str = text.Pop(2)        // "ef"; text is "abcd"
var head : str = text.PopFirst(2)   // "ab"; text is "cd"
var last : char = text.Pop()        // 'd'; text is "c"
var first : char = text.PopFirst()  // 'c'; text is empty
var empty : str = text.Pop(10)      // "": count form accepts an empty string
```

Performance: Both forms of `.PopFirst()` shift all remaining text bytes to the beginning of the string. This can take significant time for large strings, especially when repeatedly removing one byte at a time.

2.9.7.9 Overlapping Source and Destination

Dynamic string mutation preserves the logical source when it aliases the destination, including a view into the same string across a reallocation.

```
var text : str = "abcd"
text.Insert(2, text[1:3]) // "abbccd": inserts the original "bc"
text.Add(text)            // "abbccdabbccd"
```

2.9.8 Managing Dynamic Storage

2.9.8.1 .Reserve

```
.Reserve(capacity : uint)
```

`.Reserve(capacity)` ensures space for at least the specified number of text bytes without changing the length or contents. It does not reduce existing capacity. Reserving an estimated final size avoids repeated reallocations and copying while assembling text. The runtime doubles capacity as needed when a growing string exceeds its current allocation.

```
var text : str
text.Reserve(32)
text.Add("Hello, ")
text.AddFmt("{}!", ["world"]) // "Hello, world!"
```

2.9.8.2 .SetCapacity and .Compact

```
.SetCapacity(capacity : uint)
.Compact()
```

`.SetCapacity(capacity)` sets the capacity to the requested number of text bytes. Reducing it below `.length` also truncates the contents. `.SetCapacity(0)` releases the storage. `.Compact()` sets capacity to the current length. Call these after building text when reducing retained storage is useful.

```
var text : str = "abcdef"
text.SetCapacity(4)       // "abcd"; capacity is 4
text.Reserve(100)
text.Compact()            // "abcd"; capacity is 4 again
```

2.9.8.3 .Clone

```
.Clone() -> str
```

`.Clone()` returns an independent copy immediately. Ordinary `str` assignment shares storage and detaches on mutation; use `.Clone()` when the copy should already have its own storage before either value changes.

```
var original : str = "abc"
var copy : str = original.Clone()
copy[0] = 'X'             // copy is "Xbc"; original is "abc"
```

2.9.9 String Utility Methods

These methods work on every string type. They do not modify the original string. Needles, prefixes, suffixes, trim sets, and padding accept text or a `char` byte. Searches compare bytes exactly, with case sensitivity and no Unicode normalization. Case conversion, trimming, and padding return a new `str`; assign the result to keep it.

2.9.9.1 .Upper and .Lower

```
.Upper() -> str
.Lower() -> str
```

`.Upper()` and `.Lower()` return a copy with ASCII letters converted to uppercase or lowercase. They preserve non-ASCII bytes, including UTF-8 characters, and do not modify the original string. They are equivalent to `StrUpper(text)` and `StrLower(text)` and require dynamic-string support.

```
var text : embstr(32) = "Hello, é!"
var upper : str = text.Upper() // "HELLO, é!"
var lower : str = text.Lower() // "hello, é!"; text is still "Hello, é!"
```

2.9.9.2 .ToInt and .ToFloat

```
.ToInt() -> int
.ToInt(defvalue : int) -> int
```

```
.ToFloat() -> float
.ToFloat(defvalue : float) -> float
```

These methods parse a number using the same rules as StrToInt and StrToFloat. Without a fallback, invalid or empty text raises `EStrError`; with `defvalue`, conversion errors return that value. Integer conversion also checks the range of `int`. Neither method skips surrounding whitespace. Both work without dynamic-string support.

```
var integer : embstr(16) = "-42"
var count : int = integer.ToInt()        // -42
var decimal : rostr = "1,25e2"
var value : float = decimal.ToFloat()    // 125
var invalid : strslice = "invalid"
var fallback : int = invalid.ToInt(-1)   // -1
var default_value : float = invalid.ToFloat(0) // 0
var padded : rostr = " 42 "
var trimmed : int = padded.Trim().ToInt() // 42
```

2.9.9.3 .IndexOf and .LastIndexOf

```
.IndexOf(needle : strslice, start : int = 0) -> int
.IndexOf(needle : char, start : int = 0) -> int
.LastIndexOf(needle : strslice) -> int
.LastIndexOf(needle : char) -> int
```

`.IndexOf(needle, start = 0)` returns the first matching byte offset at or after `start`. `.LastIndexOf(needle)` searches for the final occurrence in the entire text. Both return `-1` when no match exists.

`.IndexOf` clamps `start` to `0 .. text.length`. An empty needle matches at the normalized start position; `.LastIndexOf("")` returns `text.length`.

```
var text : rostr = "abcdefabc"
var first : int = text.IndexOf("abc")      // 0
var next : int = text.IndexOf("abc", 1)    // 6
var last : int = text.LastIndexOf("abc")   // 6
var byte : int = text.IndexOf('d')         // 3
var missing : int = text.IndexOf("xyz")    // -1
var end : int = text.IndexOf("", 100)      // 9
```

2.9.9.4 .Contains, .StartsWith, and .EndsWith

```
.Contains(needle : strslice) -> bool
.Contains(needle : char) -> bool
.StartsWith(prefix : strslice) -> bool
.StartsWith(prefix : char) -> bool
.EndsWith(suffix : strslice) -> bool
.EndsWith(suffix : char) -> bool
```

`.Contains(needle)` tests for a match anywhere. `.StartsWith(prefix)` and `.EndsWith(suffix)` test only the corresponding end. An empty needle, prefix, or suffix matches every text value, including empty text.

```
var path : embstr(32) = "docs/reference.md"
var found : bool = path.Contains("reference") // true
var prefix : bool = path.StartsWith("docs/")  // true
var suffix : bool = path.EndsWith(".md")      // true
var case_match : bool = path.EndsWith(".MD")  // false
```

2.9.9.5 .Trim, .LTrim, and .RTrim

```
.Trim() -> str
.Trim(set : strslice) -> str
.Trim(set : char) -> str
.LTrim() -> str
.LTrim(set : strslice) -> str
.LTrim(set : char) -> str
.RTrim() -> str
.RTrim(set : strslice) -> str
.RTrim(set : char) -> str
```

`.Trim()` removes ASCII whitespace from both ends; `.LTrim()` removes it only from the left, and `.RTrim()` only from the right. Default whitespace is space, tab, line feed, carriage return, vertical tab, and form feed.

An explicit argument is a **set of bytes**, not a substring pattern. Bytes in that set are removed repeatedly from the selected ends. Interior bytes remain.

```
var text : strslice = "  abc  "
var both : str = text.Trim()       // "abc"
var left : str = text.LTrim()      // "abc  "
var right : str = text.RTrim()     // "  abc"
var marked : rostr = "-=-abc-=-"
var name : str = marked.Trim("-=") // "abc"
var dashes : str = "---abc---"
dashes = dashes.Trim('-')          // "abc": assign to replace the original
```

2.9.9.6 .LPad and .RPad

```
.LPad(length : int, fill : strslice) -> str
.LPad(length : int, fill : char) -> str
.RPad(length : int, fill : strslice) -> str
.RPad(length : int, fill : char) -> str
```

`.LPad(length, fill)` and `.RPad(length, fill)` return text padded to the requested byte length. They repeat the fill value and truncate the last repetition as needed. If the requested length is no greater than the source length, the result has the original contents. An empty fill value raises a runtime error. Padding operates on bytes and can split a multibyte UTF-8 fill value.

```
var code : rostr = "abc"
var left : str = code.LPad(8, "01") // "01010abc"
var right : str = code.RPad(5, '.') // "abc.."
var same : str = code.LPad(2, ' ')  // "abc": no truncation
```

2.9.10 Byte Indexing and Slicing

`text[index]` reads one `char` byte from any string type. `str` and embedded strings also permit replacing a byte; `rostr` and `strslice` are read-only. Indices are zero-based. An index outside the existing bytes raises a runtime range error. Negative indices do not count backward from the end; use `text[$last]` or `text[$end - 1]` for the last byte. These operations do not decode or validate UTF-8.

`text[start:end]` borrows a half-open byte range as `strslice`, while `text[start::end]` includes the end byte. Slicing is supported on every string type: `str`, `rostr`, `strslice`, `embstr(N)`, and unsized `embstr`. Slice bounds are clamped to `[0, length]`; reversed normalized bounds produce an empty slice.

`$last` is the last existing byte index and `$end` is the one-past-end position for the string being indexed or sliced. Omitting a slice bound selects the beginning or end. Negative slice bounds are clamped, rather than counted from the end; use `$end - count` to select a suffix.

```
var text : str = "abcdef"
var first : char = text[0]             // 'a'
text[0] = 'A'                          // text is "Abcdef"
var last : char = text[$last]          // 'f'
var part : strslice = text[1:3]        // "bc": end byte excluded
var inclusive : strslice = text[1::3]  // "bcd": end byte included
var head : strslice = text[:2]         // "Ab"
var tail : strslice = text[$end - 2:]  // "ef"
var all : strslice = text[-5:100]      // "Abcdef": bounds clamped
var empty : strslice = text[4:2]       // ""
var buffer : embstr(16) = "abcdef"
var from_buffer : strslice = buffer[1:4] // "bcd": borrows embedded storage
```

Slices borrow the source bytes. Keep the owner alive and its storage and contents stable while using them. Assign a slice to `str` to make an owned copy.

2.9.11 Unicode Operations

The following members decode UTF-8 and report a runtime encoding error for malformed input. A displayed symbol can combine multiple Unicode characters, for example a letter and a combining accent. `.wclen` and `.wchar[...]` are available on every string type, including `embstr(N)` and unsized `embstr`.

2.9.11.1 Unicode Character Count

`.wclen` counts decoded Unicode characters. Unlike `.length`, it requires UTF-8 decoding rather than reading a stored byte count.

```
var text : embstr(16) = "Aé€"
var bytes : int = text.length // 6
var characters : int = text.wclen // 3
```

2.9.11.2 Unicode Indexing and Slicing

`.wchar[index]` reads one Unicode character as `wchar`. `.wchar[start:end]` returns a decoded range as `[*]wchar`; the `start::end` form includes the final character. In this context, `$last` and `$end` refer to Unicode character positions. Slice bounds are clamped; an invalid character index raises a runtime range error.

```
var text : embstr(16) = "Aé€"
var second : wchar = text.wchar[1]          // 'é'
var chars : [*]wchar = text.wchar[1:$end]   // ['é', '€']
var part : str = StrFromWchars(chars[:])   // "é€": encode the selected characters
```

2.9.11.3 .ToWchars

```
.ToWchars() -> [*]wchar
```

`.ToWchars()` decodes the whole string into `[*]wchar`. It works on every string type, including `strslice`, `embstr(N)`, and unsized `embstr`. For repeated indexed Unicode processing, convert once and index the resulting array to avoid decoding the UTF-8 prefix for every lookup.

```
var text : embstr(16) = "Aé€"
var chars : [*]wchar = text.ToWchars()
var second : wchar = chars[1] // 'é'
var view : strslice = text
var from_view : [*]wchar = view.ToWchars() // ['A', 'é', '€']
```

2.10. Text Formatting

Text formatting inserts values into a string at placeholders such as `{}`. The format string supplies the surrounding text; a separate argument list supplies the values. DQ uses this text formatting syntax in multiple places, such as printing or creating exception messages.

DQ aims to be compatible with the brace-based formatting syntax of C++ `std::format` and Python `str.format()`. This chapter describes the currently supported placeholders and options.

2.10.1 Compile-Time String Interpolation

Inside a string literal, `\{expression}` inserts a constant value. Add a colon and a format specification to control its presentation:

```
const NUM : int = 123
const STR : rostr = "\{NUM:5} times..." // "  123 times..."
const HEX : rostr = "\{NUM:04X}"        // "007B"
const LABEL : rostr = "\{DQ.PKGNAME:<30}"
const PATH : rostr = "\{DQ.PKGNAME}/arma/rk/impl/hwuart_rk"
```

Expressions can contain arithmetic, named constants, nested parentheses and strings, and compile-time intrinsics. Supported values are integers, floats, booleans, characters, enums (formatted as their ordinal value), and constant text. Runtime variables and ordinary function calls are rejected, even when the enclosing string is used at runtime.

The compiler uses C++ `std::vformat` for these specifications: for example, `5` sets a minimum width, `<30` left-aligns, `04X` zero-pads hexadecimal output, and `.2f` selects two fractional digits. DQ's `u` unsigned-decimal spelling is also accepted. Characters default to text, including UTF-8 for Unicode characters; numeric specifications format their character codes. Invalid specifications produce compile errors. Width and precision must be literal parts of the specification; nested replacement fields are not supported.

Interpolation works in single-, double-, and triple-quoted literals. Ordinary braces remain literal; `\\{` writes a literal `\{`. Inserted text is not parsed again for escapes or interpolation. Multiline indentation is removed from the source text, preserving whitespace in inserted values.

The result is one compile-time literal, usable in `const` declarations and module paths. It requires no runtime formatting or dynamic strings and works with `--no-dynstrings`. These interpolations apply to expression strings; quoted filenames in directives such as `#include` retain their literal syntax.

2.10.2 Compiler Properties

`DQ` is a compiler-provided object with read-only compile-time properties. It has no runtime storage, requires no import, and cannot be shadowed by a user declaration. Property names are case-sensitive and do not take parentheses.

Property	Value
`DQ.PKGNAME`	Configured package containing the current source file, without a trailing slash
`DQ.FILENAME`	Source filename including its extension, without directories
`DQ.FILEPATH`	Absolute, lexically normalized source-file path
`DQ.FILEDIR`	Parent directory of that source file
`DQ.LINE`	One-based source line of the `DQ` token
`DQ.COLUMN`	One-based UTF-8 byte column of the `DQ` token, matching compiler diagnostics
`DQ.FUNCNAME`	Unqualified name of the enclosing function or method

Properties are evaluated where they appear. Included fragments use their own source file; returning from an include restores the enclosing source context. Assigning a property to an ordinary constant captures its value at that declaration, even when that constant is later imported by another module.

`DQ.PKGNAME` requires membership in a configured package, and `DQ.FUNCNAME` requires an enclosing function. Using either outside its applicable context is a compile error. Other file and location properties also work in standalone source files. `DQ` itself is not a value that can be assigned or passed around.

```
const IMPLEMENTATION : rostr = "\{DQ.PKGNAME}/drivers/uart"
const LOCATION : rostr = "\{DQ.FILENAME}:\{DQ.LINE}"
```

2.10.3 A Simple Print Example

The examples use `PrintLn` from `use print`. Unless stated otherwise, they use these single-letter variables. The type-specific subsections repeat the relevant declarations so their examples can be read independently.

```
var i : int = 42
var n : uint = 255
var f : float = 3.5
var s : rostr = "Amy"
var b : bool = true
var c : char = 'A'
var w : wchar = '€'
var p : ^int = %i
```

```
PrintLn("Hello, {}!", [s]) // = "Hello, Amy!"
```

`{}` is replaced by the first value in `[s]`. The square brackets belong to the DQ argument-list expression; they are not printed. `PrintLn` adds a newline after the result. Multiple placeholders consume values from left to right:

```
PrintLn("{} has {} messages.", [s, i]) // = "Amy has 42 messages."
```

2.10.4 Formatting Arguments: `varargs`

To print formatted text to the standard output use these functions from the `print` package:

```
Print(fmt : strslice, args : varargs)
PrintLn(fmt : strslice, args : varargs)
```

`fmt` is the format string. `args` is a borrowed list of values and their types.

2.10.4.1 Writing the Argument List

Group multiple values in brackets and separate them with commas. The values can have different types, and the caller evaluates them from left to right.

```
PrintLn("name={}, count={}, active={}", [s, i, b]) // = "name=Amy, count=42, active=true"
```

Each element can also be an expression:

```
PrintLn("next={}, doubled={}", [i + 1, i * 2]) // = "next=43, doubled=84"
```

Pass one value directly or in a one-element list. Use `[]` for an empty list. The entire list occupies one argument position.

```
PrintLn("count={}", i)   // = "count=42"
PrintLn("count={}", [i]) // = "count=42"
PrintLn("{} {}", s, i)  // ERROR: too many arguments
```

2.10.4.2 Which Values Can Be Passed?

Value type	Example argument	Default output with `{}`
Signed integers	`-7`, an `int32` or `int64` variable	Signed decimal number
Unsigned integers	A `uint` or `uint64` variable	Unsigned decimal number
Floating-point numbers	`3.5`, a `float32`, `float64`, or `float` variable	Decimal number
Boolean values	`true`, `false`, a comparison	`true` or `false`
`char`	A `char` variable	One byte written as a character
`wchar`	`'€'`, a `wchar` variable	One Unicode character encoded as UTF-8
`char16`	A `char16` variable	One Unicode character; converted to `wchar`
Text	A literal, `str`, `rostr`, `strslice`, `embstr(N)`, or unsized `embstr`	The text bytes
`^char`	A pointer to zero-terminated text	The pointed-to text
Other typed or generic pointers	A `^int` or `pointer` variable	Address in hexadecimal with a leading `^`
Nil pointer	`nil`	`^0`

The formatter preserves the value's type. A text value containing `"42"` is still text, and an integer containing `42` is still a number. The placeholder can choose a representation, but it does not perform arbitrary conversions between these categories.

```
PrintLn("{} / {} / {} / {}", [i, f, b, w]) // = "42 / 3.5 / true / €"
```

Objects, structures, function references, and whole arrays cannot be passed as values for automatic formatting. Select their fields or elements instead.

```
var a : [3]int = [10, 20, 30]

PrintLn("first={}, count={}", [a[0], a.length]) // = "first=10, count=3"
```

For a stored `anyvalue`, pass a value read with an accessor such as `AsInt` or `AsText`.

2.10.4.3 Forwarding Arguments

A function can pass its `varargs` parameter directly to another formatter. The list is valid throughout the original call.

```
func Log(fmt : strslice, args : varargs):
    PrintLn(fmt, args)
endfunc

Log("{} has {} messages.", [s, i]) // = "Amy has 42 messages."
```

2.10.4.4 Matching Values to Format String Placeholders

Each placeholder uses the next value. To print the same value multiple times, include it in the `args` multiple times:

```
PrintLn("{} + {} = {}", [i, i, i + i]) // = "42 + 42 = 84"
```

One argument is used for each placeholder. There are no named or numbered argument references. The current formatter writes `null` for a missing argument and ignores arguments left over after the last placeholder:

```
PrintLn("{} / {}", [s]) // = "Amy / null"
PrintLn("{}", [s, i])   // = "Amy"
```

These mismatches do not generate compile-time or runtime errors.

2.10.5 Format String Placeholders

Ordinary characters in the format string are copied to the output. Each `{...}` placeholder inserts the next argument. `{}` uses the default representation; a colon introduces formatting options.

2.10.5.1 Main Placeholder Variants

Form	Accepted values	Meaning
`{}` / `{:?}`	Any supported value	Default representation. Use this for `bool` types
`{:d}`	Integers, characters	Signed decimal number or character code
`{:u}`	Integers, characters	Unsigned decimal number or character code
`{:x}`	Integers, characters	Lowercase hexadecimal
`{:X}`	Integers, characters	Uppercase hexadecimal
`{:p}`	Pointer values	Hexadecimal address with a leading `^`
`{:s}`	Text, characters	Text bytes or a character
`{:f}`	Floating-point values	Decimal floating-point output
`{{`	No argument	Literal `{`
`}}`	No argument	Literal `}`

2.10.5.2 Signed Integers: `{:d}`

Use `{:d}` for signed decimal output. `{}` gives the same result for a signed integer.

```
var i : int = 42

PrintLn("{:d}", [i])     // = "42"
PrintLn("[{:6d}]", [i])  // = "[    42]": minimum width
PrintLn("{:06d}", [i])   // = "000042": zero padding
PrintLn("{:+d}", [i])    // = "+42": explicit sign
PrintLn("{:+06d}", [i])  // = "+00042": sign followed by zero padding
PrintLn("[{: d}]", [i])  // = "[ 42]": reserve a sign position
PrintLn("{:06d}", [-i])  // = "-00042": negative sign before the zeros
PrintLn("[{:<6d}]", [i]) // = "[42    ]": left alignment
```

2.10.5.3 Unsigned Integers: `{:u}`

Use `{:u}` for unsigned decimal output, including the full range of `uint`.

```
var n : uint = 255

PrintLn("{:u}", [n])     // = "255"
PrintLn("[{:6u}]", [n])  // = "[   255]"
PrintLn("{:06u}", [n])   // = "000255"
PrintLn("[{:<6u}]", [n]) // = "[255   ]"
```

2.10.5.4 Hexadecimal Integers: `{:x}` and `{:X}`

Use `x` for lowercase digits and `X` for uppercase digits. These formats accept signed and unsigned integers.

```
var n : uint = 255

PrintLn("{:x}", [n])   // = "ff"
PrintLn("{:X}", [n])   // = "FF"
PrintLn("{:08X}", [n]) // = "000000FF": zero-padded hexadecimal
PrintLn("0x{:x}", [n]) // = "0xff": add the prefix as literal text
```

2.10.5.5 Floating-Point Numbers: `{:f}`

Use `{:f}` for decimal floating-point output.

```
var f : float = 3.5

PrintLn("{:f}", [f])       // = "3.5"
PrintLn("{:.2f}", [f])     // = "3.50": two fractional digits
PrintLn("{:.0f}", [f])     // = "4": rounded to a whole number
PrintLn("[{:8.2f}]", [f])  // = "[    3.50]": width includes the decimal point
PrintLn("{:+.2f}", [f])    // = "+3.50"
PrintLn("{:+08.2f}", [f])  // = "+0003.50"
PrintLn("[{:<8.2f}]", [f]) // = "[3.50    ]"
```

An explicit precision fixes the number of digits after the decimal point, rounding as needed and retaining trailing zeros. The decimal separator is a dot. Without explicit precision, `{}` and `{:f}` use up to 15 fractional digits and remove trailing zeros. Explicit precision is limited to 18 fractional digits.

The `f` placeholder requires a floating-point value; it does not convert an integer to a float.

2.10.5.6 Text: `{:s}`

Use `{:s}` or `{}` for text. The same formats accept the other supported string types and `^char`. Here, `t` contains ten characters.

```
var s : rostr = "Amy"
var t : rostr = "abcdefghij"

PrintLn("{:s}", [s])     // = "Amy"
PrintLn("[{:8s}]", [s])  // = "[     Amy]": right alignment is the default
PrintLn("[{:<8s}]", [s]) // = "[Amy     ]": left-aligned text column
PrintLn("[{:>8s}]", [s]) // = "[     Amy]": explicit right alignment
PrintLn("[{:^8s}]", [s]) // = "[  Amy   ]": centered text
PrintLn("[{:8s}]", [t])  // = "[abcdefghij]": longer text is not truncated
```

The `s` kind is optional: `{:8}` and `{:8s}` both reserve a right-aligned field at least eight Unicode code points wide. Width never truncates longer text.

2.10.5.7 Booleans: `{}`

Use `{}` or `{:?}` for `bool`; there is no separate Boolean format letter.

```
var b : bool = true

PrintLn("{}", [b])     // = "true"
PrintLn("{}", [not b]) // = "false"
PrintLn("[{:6}]", [b]) // = "[  true]"
```

2.10.5.8 Characters: `{:s}`

Default formatting and `{:s}` write a character. Explicit numeric formats write its code.

```
var c : char  = 'A'
var w : wchar = '€'

PrintLn("{}", [c])     // = "A"
PrintLn("{}", [w])     // = "€"
PrintLn("{:s}", [w])   // = "€"
PrintLn("{:d}", [c])   // = "65"
PrintLn("{:u}", [w])   // = "8364"
PrintLn("{:04X}", [c]) // = "0041"
```

A `char` writes one byte; a `wchar` is encoded as UTF-8. A `char16` value is converted to `wchar` before formatting. A UTF-16 surrogate pair is not combined automatically when its two code units are passed as two arguments. Convert UTF-16 text with `StrFromUtf16()` first. An invalid Unicode character value, including an isolated surrogate, produces `�`.

2.10.5.9 Pointers: `{:p}`

Use `{:p}` or `{}` for an address in hexadecimal with a leading `^`. `%i` takes the address of `i`. The output below uses an example address; the actual address varies between runs.

```
var i  : int = 42
var p  : ^int = %i
var z  : pointer = nil
var pc : ^char = "abc"

PrintLn("{:p}", [p])           // = "^7FFC1234ABCD": address of i
PrintLn("{}", [p])             // = "^7FFC1234ABCD": same address with default formatting
PrintLn("{:p}", [z])           // = "^0": nil pointer
PrintLn("{}", [pc])            // = "abc": ^char is treated as text
PrintLn("{:p}", [pointer(pc)]) // = "^401234": convert to pointer to print the address
```

2.10.5.10 Missing Arguments and Nil Pointers

A missing argument prints `null`. A supplied nil pointer prints `^0`.

```
PrintLn("{} / {}", nil) // = "^0 / null"
```

2.10.5.11 Literal Braces and Escaping

Double each brace in a format string to print one literal brace. Escaped braces do not consume arguments. In a double-quoted string literal, escape embedded double quotes as `\"`:

```
var s : rostr = "Amy"

PrintLn("{{{}}}", [s])              // = "{Amy}"
PrintLn("{{\"name\":\"{}\"}}", [s]) // = '{"name":"Amy"}': escaped double quotes
PrintLn('{{"name":"{}"}}', [s])     // = '{"name":"Amy"}': single outer quotes avoid internal escaping
PrintLn("{{}}", [])                 // = "{}": just the literal braces
PrintLn("{}")                       // = "{}": no doubling need at non-formatting print
```

2.10.6 Width, Fill, Sign, Precision

The type-specific examples above show the common combinations. The supported option order is:

```
{: [alignment] [sign] [0] [width] [.precision] [kind] }
```

Spaces and square brackets in this diagram are explanatory; do not write them in the placeholder. Each bracketed part is optional.

Option	Example	Meaning
`<`	`{:<8s}`	Left-align within the field
`>`	`{:>8s}`	Right-align within the field
`^`	`{:^8s}`	Center within the field
`+`	`{:+d}`	Show `+` for a non-negative signed number or float
`-`	`{:-d}`	Show a sign only for negative values; this is the default
Space	`{: d}`	Use a leading space for a non-negative signed number or float
`0`	`{:06d}`	Use zeros for right-aligned numeric padding
Width	`{:8s}`	Reserve a field at least eight Unicode code points wide
Precision	`{:.2f}`	Write two digits after the decimal point

Width and alignment work with text, characters, numbers, Boolean values, null, and pointers. DQ defaults to **right alignment for every value type**. `<` pads on the right, `>` pads on the left, and `^` divides padding between both sides. If centering needs an odd number of spaces, the extra space goes on the right.

Width is a minimum, not a maximum: longer values are never truncated to fit. Width counts **Unicode code points**, not UTF-8 bytes. For example, `€` counts as one code point even though its UTF-8 encoding uses three bytes, and `café` counts as four:

```
var w : wchar = '€'
var s : rostr = "café"

PrintLn("[{:>5s}]", [w]) // = "[    €]": four spaces plus one code point
PrintLn("[{:>6s}]", [s]) // = "[  café]": right alignment
PrintLn("[{:<6s}]", [s]) // = "[café  ]": left alignment
PrintLn("[{:^6s}]", [s]) // = "[ café ]": centered text
```

Combining marks count separately. Width does not measure visual character groups or terminal display columns.

A negative sign is written automatically. `+` also requests a sign for non-negative signed numbers and floating-point values. A space sign option reserves a sign position without writing `+`. The `-` option requests the default sign behavior; use `<` for left alignment.

Zero padding applies to right-aligned numbers and is written after the sign or pointer prefix. Left and center alignment use spaces even if `0` is present. Text and character output also use spaces.

Write alignment first, followed by the sign, zero fill, and width. For example, `{:^+8d}` centers a signed number, and `{:+08d}` enables zero fill with a sign. The former zero-before-sign spelling `{:0+8d}` is no longer supported.

2.10.7 Formatting APIs and Destinations

The same placeholders and `varargs` argument syntax work with each API:

API	Destination	Availability
`Print(fmt, args)` / `PrintLn(fmt, args)`	Standard output	`use print`
`Format(fmt, args)`	A new `str`	Implicit `sys`; requires dynamic-string support
`.AddFmt(fmt, args)` on `str`	Appends to a dynamic string	Requires dynamic-string support
`.AddFmt(fmt, args)` on `embstr`	Appends to a fixed buffer	Works without dynamic-string support
`Exception(fmt, args)`	Exception message	Requires exception support

The `fmt` parameter is `strslice`: literals, `str`, `rostr`, `embstr`, and existing slices can be passed directly. Even a substring can be used as the format:

```
var i : int = 42
var s : rostr = "{} ignored"

PrintLn(s[0:2], [i]) // = "42": only the "{}" slice is parsed
```

2.10.7.1 Print and PrintLn

```
Print(text : strslice)
Print(fmt : strslice, args : varargs)
PrintLn()
PrintLn(text : strslice)
PrintLn(fmt : strslice, args : varargs)
```

`Print` writes without adding a newline. `PrintLn` adds a newline, and `PrintLn()` writes a blank line.

```
Print("Hello, {}", [s]) // = "Hello, Amy": without a newline
PrintLn("!")            // = "!": followed by a newline
PrintLn()               // = "": a blank line
```

The one-argument form writes its text **literally**, without interpreting placeholders or escaped braces. Supplying the second argument selects the formatting overload:

```
PrintLn("{}")       // = "{}"
PrintLn("{{}}")     // = "{{}}"
PrintLn("{{}}", []) // = "{}"
```

2.10.7.2 Format

```
Format(fmt : strslice, args : varargs) -> str
```

Use `Format()` when the result should be stored or returned rather than immediately printed. `Format()` is included in the implicit `sys`, which means it is available without a `use` declaration.

```
var s : str = Format("item-{:04d}", [i]) // = "item-0042": contents of s
```

`Format()` is also useful when passing formatted text to a function that does not support formatting itself. For example, a database connection's `.Execute(sql)` method takes a complete SQL statement. Here, `dbconn` is an open database connection and `items` is a table with one integer column:

```
var i : int = 42
dbconn.Execute(Format("INSERT INTO items VALUES ({:d})", [i]))
```

2.10.7.3 .AddFmt

```
.AddFmt(fmt : strslice, args : varargs)
```

`.AddFmt()` appends to the existing contents of the `str` or `embstr`.

`str` and `embstr(n)` declarations without initializers initialize the string variables empty, so `.AddFmt()` can be called immediately (calling `.Clear()` before is not necessary):

```
var ds : str
ds.AddFmt("Order {}: {} items", [i, n]) // = "Order 42: 255 items": contents of s
var es : embstr(32)
es.AddFmt("mask = {:04X}", [n]) // = "mask = 00FF": contents of s
```

A `str` grows as needed by re-allocating its storage memory. An `embstr(N)` keeps its fixed buffer, including space for the terminating zero, so formatting can truncate the result:

```
var es : embstr(8)
es.AddFmt("{}", ["abcdefghij"]) // = "abcdefg": contents of s; seven bytes plus the terminator fit
```

Repeated `.AddFmt()` calls can assemble a larger message. Reserve space with `.Reserve()` first when the expected size of a dynamic string is known. This avoids repeated buffer growth and can reduce heap fragmentation.

The following example reserves 50000 bytes, enough for the numbers 1 through 10000, each followed by a space:

```
use print

func AppendNumbers(s : ref str):
    for i : int = 1 to 10000:
        s.AddFmt("{:d} ", [i])
    endfor
endfunc

func *Main():
    var s : str
    s.Reserve(50000)
    AppendNumbers(s)
    PrintLn("String length: {}", [s.length]) // = "String length: 48894"
endfunc
```

2.10.7.4 Exception Messages

The text formatting can be used at rising exceptions too:

```
var i : int = -42
if i < 1:
    raise Exception("invalid count: {}", [i]) // = "invalid count: 42"
endif
```

See Exceptions and Error Handling for raising, catching, and inspecting exceptions.

2.10.8 Argument and Error Rules

Formatting uses the stored value's type. For example, `{:f}` requires a float, while `{:d}` expects an integer or a character code. Convert the value explicitly before formatting when another category is needed:

```
PrintLn("{:.2f}", [float(i)])     // = "42.00"
PrintLn("{:d}", [StrToInt("42")]) // = "42"
```

A type/format mismatch produces a diagnostic marker instead of throwing a formatting exception:

```
PrintLn("{:f}", [i]) // = "<i-f!>": an integer with a float placeholder
PrintLn("{:d}", [s]) // = "<s-f!>": text with a numeric placeholder
PrintLn("{:d}", [b]) // = "<b-f!>": a Boolean with a numeric placeholder
```

2.10.8.1 Low-Level Formatter Flags

`Format()`, `.AddFmt()`, and the print helpers do not return a formatting status. For direct control, `rtl/textformat` exposes `OTextFormatter`, which writes through an `OTextSink` such as `OTextSinkEmbStr`, `OTextSinkStr`, or `OTextSinkStdOut`. Its `.flags` records `DQFMT_FMT_ERROR` for malformed braces or unsupported format options, and `DQFMT_ARG_ERROR` for incompatible value types or invalid Unicode characters. Each formatting call resets the flags.

With `use rtl/textformat`, let `t` be an `OTextFormatter` connected to the desired sink:

```
t.Format("{:f}", [i]) // = "<i-f!>": sets t.flags to DQFMT_ARG_ERROR
```

The runtime also defines `DQFMT_TRUNCATED` and `DQFMT_SINK_ERROR`, but the current built-in sinks do not set them when a fixed buffer fills. Keep the destination capacity in mind when complete output is required.

2.11. Arrays and Slices

An array keeps several values of the same type together, such as a list of names, a byte buffer, or a week's temperature readings. Use a **fixed array** when the number of values is known in advance, or a **dynamic array** when you need to add and remove values. A **slice** lets you work with part of an array without copying its elements.

Here is a small program that builds a shopping list of strings and prints it:

```
use print

func PrintList(list : []str):
    for item :? in list:
        PrintLn("{}", [item])
    endfor
endfunc

func *Main():
    var shopping : [*]str = ["bread", "milk"]
    shopping.Append("apples")
    shopping.Append(["potatoes", "tomatoes"])
    var i : int = shopping.IndexOf("potatoes")
    if i >= 0:
        shopping[i] = "rice"
    endif
    PrintList(shopping)
endfunc
```

This prints:

```
bread
milk
apples
rice
tomatoes
```

2.11.1 Array Forms

In the types below, `T` stands for the element type (e.g. `int`, `float`, `str` etc.), `N` is the number of elements in a fixed array.

Form	Meaning	When to use it
`[N]T`	A fixed array of `N` elements	The size is known in advance, such as seven daily readings.
`[?]T`	A fixed array whose size comes from its initial values	You want the compiler to count the values for you.
`[]T`	A slice that borrows existing elements	You want to pass an array or part of it to a function.
`[*]T`	A dynamic array that can grow or shrink	You collect values as the program runs.

Every form has a `.length` property that returns the number of the array elements.

2.11.2 Array Properties and Methods

Use this table to find an operation, then follow its link for examples. **D** means dynamic arrays, **F** means fixed arrays, and **S** means slices. A result of `—` means the method returns no value; `T` is the element type.

Property or method	Types	Result	Purpose
`.Append(value)` `.Append(values)`	D	—	Add one or more elements at the end.
`.AppendSlice(values)`	D	—	Append the elements of another array or slice.
`.capacity`	D	`int`	Space available before more storage is needed.
`.Clear(free_storage = false)`	D	—	Remove everything; optionally release storage.
`.Clone()`	D	`[*]T`	Make an independent copy.
`.Compact()`	D	—	Reduce capacity to match the length.
`.Delete(index, count = 1)`	D	—	Remove one or more elements.
`.IndexOf(value)`	D, F, S	`int`	Find the first matching element, or return `-1`.
`.Insert(index, value)` `.Insert(index, values)`	D	—	Add one or more elements before a position.
`.InsertSlice(index, values)`	D	—	Insert the elements of another array or slice.
`.length`	D, F, S	`int`	Number of elements.
`.Pop()`	D	`T`	Remove and return the last element.
`.PopFirst()`	D	`T`	Remove and return the first element.
`.Prepend(value)` `.Prepend(values)`	D	—	Add one or more elements at the beginning.
`.refcount`	D	`int`	Number of references sharing the array.
`.Reserve(capacity)`	D	—	Make room for later additions.
`.SetCapacity(capacity)`	D	—	Set storage size, removing excess elements if needed.
`.SetLength(length)`	D	—	Change the number of elements.

2.11.3 Fixed Arrays

Use a fixed array when the number of elements will stay the same. Write the size inside brackets before the element type:

```
var readings : [7]float = {}              // seven readings, initially all zero
var header : [3]byte = [0x44, 0x51, 0x01] // exactly three bytes
var colors : [?]str = ["red", "green", "blue"] // the compiler infers [3]str
const lookup : [?]uint8 = [0x00, 0x29, 0x3B] // the compiler infers [3]uint8
```

`[?]T` works for variables and constants and needs a nonempty array literal as its initializer so the compiler can count the elements. The length is part of a fixed array's type: `[3]int` and `[4]int` are different types. You can replace individual elements in array variables, but cannot add or remove any.

Assigning a fixed array to another compatible fixed array copies its elements:

```
var original : [3]float = [18.5, 21, 19.5]
var copy : [3]float = original
copy[0] = 22.5 // original[0] is still 18.5
```

Fixed arrays store their elements directly with the variable or containing structure. For arrays of `autofree` elements, assignment first frees the destination's previous objects, then transfers each source element and sets the source elements to `nil`. Assigning an array to itself leaves it unchanged. See the automatic cleanup rules.

2.11.4 Dynamic Arrays

Use `[*]T` when the number of elements can change. Dynamic arrays are initialized empty at the declaration, you can append elements using the `.Append()` method:

```
var names : [*]str
names.Append("Amy")
names.Append("Lin")
names.Append("Sam") // names is ["Amy", "Lin", "Sam"]
```

Assigning a dynamic array to another variable shares the same array. Changes to the elements, length, or capacity are visible through both variables. Use `.Clone()` when you need a separate copy:

```
var names : [*]str = ["Amy", "Lin"]
var shared : [*]str = names
shared[0] = "Sam"     // names is now ["Sam", "Lin"]
shared.Append("Jo")   // names is now ["Sam", "Lin", "Jo"]

var copy : [*]str = names.Clone()
copy[0] = "Alex"      // names[0] is still "Sam"
```

Assigning a different array to `shared` changes which array that variable uses; it does not make `names` use the new array.

Dynamic arrays allocate their element storage as needed and manage its lifetime automatically. An empty array can exist without allocating any storage.

Arrays of managed types, such as `str`, need extra runtime handling when elements are copied or removed. The compiler adds this handling automatically.

2.11.5 Indexing

Use `array[index]` to read or replace one element. Positions start at **zero**: the first element is at `0`, the second at `1`, and so on.

```
var names  : [*]str = ["Amy", "Lin", "Sam"]
var first  : str = names[0]    // "Amy"
var second : str = names[1]    // "Lin"
names[1] = "Jo"                // names is now ["Amy", "Jo", "Sam"]
var last : str = names[$last]  // "Sam"
```

`$last` means `length - 1`, the position of the last element. `$end` means `length`, the position just after it. Use `$end` for a slice bound or to insert at the end; it does not select an existing element.

Make sure the array is nonempty before using `$last`, and keep element indices between `0` and `length - 1`. Negative indices do not count backward from the end: use `array[$last]` or `array[$end - 1]` for the last element. Indexing does not adjust an invalid index to fit the available range.

Dynamic arrays report a runtime range error for invalid indices. Fixed arrays and slices currently omit runtime bounds checks for speed. An invalid access can read unrelated memory, corrupt data, or crash the program.

2.11.6 Slices

A slice is a view of existing elements. Use one to process just part of an array or to pass arrays of different sizes to the same function. Write `[]T` for its type and `array[start:end]` to select a range. Slicing works on fixed arrays, dynamic arrays, and existing slices, with the same range rules as strings.

The start position is included and the end position is excluded. For example, `data[1:3]` selects elements `1` and `2`:

```
var data   : [*]byte = [0x10, 0x20, 0x30, 0x40]
var middle : []byte  = data[1:3]  // [0x20, 0x30]
middle[0] = 0xFF                  // data is now [0x10, 0xFF, 0x30, 0x40]
```

Changing a slice element changes the original array because no elements were copied. A slice has its own length, but cannot add or remove elements from its source array.

Omit a bound to select from the beginning or through the end. Use `::` instead of `:` when you want to include the element at the end position:

```
var data : [*]byte = [0x10, 0x20, 0x30, 0x40]
var all  : []byte = data[:]         // [0x10, 0x20, 0x30, 0x40]
var head : []byte = data[:2]        // [0x10, 0x20]
var tail : []byte = data[2:]        // [0x30, 0x40]
var last_two  : []byte = data[$end - 2:] // [0x30, 0x40]
var inclusive : []byte = data[1::3] // [0x20, 0x30, 0x40]
```

Slice bounds are limited to the available range: a negative bound becomes zero and a bound past the end becomes `length`. Negative slice bounds do not count backward from the end. If the resulting end is before the start, the slice is empty.

```
var data   : [4]int = [10, 20, 30, 40]
var all    : []int  = data[-5:100] // [10, 20, 30, 40]
var empty  : []int  = data[3:1]  // no elements
var at_end : []int  = data[$end:] // no elements
var negative_end : []int  = data[:-1] // no elements; -1 becomes zero
```

A slice borrows its source; it does not keep that source alive. Keep the original array alive while using the slice. Adding or removing elements, resizing, or releasing a dynamic array's storage can make existing slices and element pointers invalid. Create a new slice after such a change, even if the change seems to affect another part of the array. You also cannot keep a slice of a temporary array that disappears before the slice does.

2.11.7 Function Parameters

Use a slice parameter, `[]T`, when a function needs to read or update elements without resizing the array. The caller can pass a fixed array, a dynamic array, or a slice with a compatible element type. Passing an array automatically gives the function a view of all its elements:

```
func Sum(values : []float) -> float:
    result = 0
    for value :? in values:
        result += value
    endfor
endfunc

func *Main():
    var fixed   : [3]float = [18.5, 21, 19.5]
    var growing : [*]float = [20.5, 22]
    var total   : float = Sum(fixed)     // 59
    var other   : float = Sum(growing)   // 42.5
    var part    : float = Sum(fixed[1:]) // 40.5
endfunc
```

You can also pass an array returned by a function directly to a slice parameter:

```
func MakeItems() -> [*]str:
    return ["a", "b"]
endfunc

func JoinItems(items : []str) -> str:
    return StrJoin(items, "|")
endfunc

// JoinItems(MakeItems()) returns "a|b".
```

The temporary array stays alive until the receiving call finishes, including when it raises an exception. A slice borrowed from that temporary must not be kept after the call returns. This also applies to returned fixed arrays.

A slice parameter can also change the caller's elements, just like the `middle` slice in the previous section. For a function that adds or removes elements, use a dynamic array passed by reference:

```
func AddName(names : ref [*]str, name : str):
    names.Append(name)
endfunc

func *Main():
    var names : [*]str = ["Amy", "Lin"]
    AddName(names, "Sam") // names is now ["Amy", "Lin", "Sam"]
endfunc
```

Dynamic-array parameters require a reference mode; `names : [*]str` by value is not supported. A `ref` parameter can resize the array or replace the caller's array variable. Other parameter modes and their read/write restrictions are explained in Functions.

2.11.8 Iteration

Use `for ... in` to work with each element in order. `:?` asks the compiler to infer the loop variable's type:

```
var readings : [*]float = [18.5, 21, 19.5]
var total    : float = 0
for reading :? in readings:
    total += reading
endfor
// total is 59
```

The ordinary loop variable receives each element's value. Use `ref` when you want to change the array elements themselves:

```
var readings : [*]float = [18.5, 21, 19.5]
for reading : ref in readings:
    reading += 0.5
endfor
// readings is now [19, 21.5, 20]
```

A `ref` loop variable refers directly to the current array element and has its exact type. Do not add or remove elements or otherwise resize the dynamic array while using that reference: the element may move or disappear.

When the array stores `autofree` elements, an ordinary loop variable refers to each object without removing it from the array or freeing it after the iteration. Use a `ref` loop variable to replace an element in place; this automatically frees the previous object. Elements can be `nil`, so check before using their fields or methods. See Dynamic Array `autofree` Elements.

2.11.9 Array Properties

Properties let you inspect an array. They are read-only; use methods such as `.SetLength()` and `.Reserve()` to change its size or storage.

2.11.9.1 `.length`

`.length` counts elements, not bytes. Use it to check whether an array is empty or to find out how many values a slice contains. It works on every array form.

```
var data   : [4]int = [10, 20, 30, 40]
var middle : []int = data[1:3]
var total  : int = data.length   // 4
var count  : int = middle.length // 2
var empty  : [*]int
var is_empty : bool = empty.length == 0 // true
```

2.11.9.2 `.capacity`

`.capacity` tells you how many elements fit in a dynamic array's current storage. It is at least as large as `.length`. Extra capacity is spare room for future additions; it does not create extra elements that you can index.

```
var scores : [*]int = [10, 20, 30]
scores.Reserve(100)
var count : int = scores.length         // 3
var has_room : bool = scores.capacity >= 100 // true
```

When growth exceeds the capacity, the array obtains more storage automatically. An empty array with no allocated storage has capacity zero.

2.11.9.3 `.refcount`: Reference Count

`.refcount` is mainly useful for checking how many references share a dynamic array. Assigning it to another variable adds a reference; creating a slice does not. Temporary values can also add references. An array with no allocated storage has reference count zero.

```
var scores : [*]int = [10, 20, 30]
var shared : [*]int = scores
var references : int = scores.refcount // 2
```

2.11.10 Dynamic Array Methods

See Array Properties and Methods for a quick overview.

Call these methods on a dynamic array, such as `names.Append("Amy")`. Methods that change the array affect every variable sharing it. `.Clone()` returns a separate array, and `.Pop()` and `.PopFirst()` return the element they remove. The other methods return no value.

The prototypes below use `T` for the array's element type: for `[*]str`, `T` means `str`. A `values : []T` parameter accepts a fixed array, a dynamic array, or a slice with a compatible element type. These prototypes describe the built-in methods; you do not need to declare them yourself.

You can pass part of the same array as the source for an append or insert. The method preserves the source values even if it needs to move the array's storage:

```
var values : [*]int = [10, 20, 30]
values.Append(values[:2]) // [10, 20, 30, 10, 20]
```

After a method changes the array's size or storage, previously created slices for this array might become invalid; see Slices. Arrays of `autofree` elements have additional move and copy restrictions; see Dynamic Array `autofree` Elements.

2.11.10.1 .Append and .Prepend

```
.Append(value : T)
.Append(values : []T)
.AppendSlice(values : []T)
.Prepend(value : T)
.Prepend(values : []T)
```

Use `.Append()` to build a list one value at a time or add a whole group at once. Use `.Prepend()` to put values at the beginning. When adding several elements, both methods keep their original order.

```
var names : [*]str = ["Lin"]
names.Append("Sam")            // ["Lin", "Sam"]
names.Prepend(["Amy", "Jo"])   // ["Amy", "Jo", "Lin", "Sam"]
names.AppendSlice(["Alex"])    // ["Amy", "Jo", "Lin", "Sam", "Alex"]
```

`.AppendSlice()` explicitly adds the source's elements. This is useful for arrays whose elements are themselves arrays, where `.Append()` may instead add the source as one element.

Prepending moves the existing elements to make room. For a large list built one element at a time, appending usually does less work.

2.11.10.2 .Insert and .InsertSlice

```
.Insert(index : int, value : T)
.Insert(index : int, values : []T)
.InsertSlice(index : int, values : []T)
```

Use `.Insert()` to put values before an existing position. Elements at that position and after it move to make room. `.InsertSlice()` explicitly inserts all the source elements, like the range form of `.Insert()`.

```
var stops : [*]str = ["home", "office"]
stops.Insert(1, ["bakery", "school"]) // ["home", "bakery", "school", "office"]
stops.Insert($end, "park")           // ["home", "bakery", "school", "office", "park"]
```

`$end` inserts after the last element, like `.Append()`. A negative insertion position inserts at the beginning, and a position past the end inserts at the end. Inserting near the beginning of a large array moves more elements than inserting near its end.

2.11.10.3 .Delete

```
.Delete(index : int, count : int = 1)
```

Use `.Delete(index)` to remove one element, or supply `count` to remove several. The elements after the removed range move to close the gap and keep their order.

```
var scores : [*]int = [10, 20, 30, 40]
scores.Delete(1, 2) // [10, 40]: remove elements at positions 1 and 2
scores.Delete(1)    // [10]
scores.Delete(100)  // [10]: nothing at this position
```

A negative position starts deleting at the beginning. A count larger than the remaining length removes everything from that position onward. A non-positive count or a position at or past the end leaves the array unchanged.

2.11.10.4 .Pop and .PopFirst

```
.Pop() -> T
.PopFirst() -> T
```

Use `.Pop()` to take the last element, for example when treating an array as a stack. Use `.PopFirst()` to take the first element, such as the next job in a small queue. Both remove the element and return its value.

```
var jobs : [*]str = ["download", "resize", "save"]
var next : str = jobs.PopFirst() // "download"; jobs is ["resize", "save"]
var last : str = jobs.Pop()      // "save"; jobs is ["resize"]
```

Check `.length` before popping if the array might be empty: both methods raise a runtime error on an empty array. `.PopFirst()` moves all remaining elements toward the beginning, so repeatedly using it on a large queue can be expensive.

2.11.10.5 .SetLength

```
.SetLength(length : uint)
```

Use `.SetLength()` when you know how many elements you need, for example before filling a byte buffer by index. Growing adds initialized elements: new numbers are zero and new strings are empty. Shrinking removes elements from the end but keeps the spare storage for later use.

```
var packet : [*]byte
packet.SetLength(3) // [0, 0, 0]; length and capacity are 3
packet[0] = 0x44
packet[1] = 0x51
packet[2] = 0x01
packet.SetLength(2) // [0x44, 0x51]; capacity stays 3
```

The requested length must be non-negative. If it exceeds the current capacity, `.SetLength()` allocates space for exactly that many elements. Existing spare capacity is retained when the requested length already fits.

When you append an element to an array with no spare space, the runtime doubles its allocated capacity, leaving room for later additions. Appending to an array with no allocated storage starts with room for one element.

2.11.10.6 .Reserve

```
.Reserve(capacity : uint)
```

Use `.Reserve()` before collecting many values when you can estimate the final size. It makes room for at least that many elements, which can avoid repeatedly moving the array into larger storage. It does not add elements, change existing values, or reduce capacity.

```
var readings : [*]float
readings.Reserve(100) // still empty, with room for at least 100 readings
```

```
readings.Append(18.5)
readings.Append(21.5) // [18.5, 21.5]; the reserved space is being used
```

Capacity must be non-negative. You can keep appending beyond the reserved amount; the array will grow automatically.

2.11.10.7 .SetCapacity and .Compact

```
.SetCapacity(capacity : uint)
.Compact()
```

Use `.Compact()` after building an array if you want to release its unused space. It reduces capacity to the current length without removing elements.

`.SetCapacity()` gives you direct control over the storage size. If the requested capacity is smaller than the length, elements at the end are removed to fit. Increasing capacity leaves the elements and length unchanged.

```
var values : [*]int = [10, 20, 30]
values.SetCapacity(2) // [10, 20]; length and capacity are 2
values.Reserve(100)
values.Compact()      // [10, 20]; capacity is 2 again
```

Capacity must be non-negative. Use `.Reserve()` when you only want to make room for later additions without risking removal of existing elements.

2.11.10.8 .Clear

```
.Clear(free_storage : bool = false)
```

Use `.Clear()` to empty a list before filling it again. It keeps the storage so later additions can reuse it. Pass `true` when you also want to release the storage. Both forms leave the array empty.

```
var messages : [*]str = ["Connected", "Ready"]
messages.Clear()             // length is 0; capacity is retained
messages.Append("Restarted") // ["Restarted"]: reuse the existing storage
messages.Clear(true)         // length and capacity are 0
```

2.11.10.9 .Clone

```
.Clone() -> [*]T
```

Use `.Clone()` to keep an independent copy before changing an array. Assigning the result to a new variable lets you edit that copy without changing the original array's elements.

```
var names : [*]str = ["Amy", "Lin", "Sam"]
var saved : [*]str = names.Clone()
names[0].Append(" Lovelace") // saved[0] is still "Amy"
```

For `str` elements, later edits to a string in either array leave the other array's string unchanged, as shown above.

Cloning copies the elements according to their types. Object references still point to the same objects, and nested dynamic arrays still share their inner arrays. Clone those inner arrays too if you need separate copies at every level. Cloning arrays of `autofree` elements is not supported; see Dynamic Array `autofree` Elements.

2.11.11 Searching and Comparison

2.11.11.1 .IndexOf

```
.IndexOf(value : T) -> int
```

Use `.IndexOf()` to find an element in a fixed array, dynamic array, or slice. It returns the first matching position, starting at zero, or `-1` if the value is absent. Check for `-1` before using the result as an index.

```
var scores : [*]int = [10, 20, 30, 20]
var position : int = scores.IndexOf(20) // 1: the first match
```

```
if position >= 0:
    scores[position] = 25             // [10, 25, 30, 20]
endif
var missing : int = scores.IndexOf(99) // -1
```

For a slice, the result counts from the start of that slice:

```
var data : [4]int = [10, 20, 30, 40]
var middle : []int = data[1:3]          // [20, 30]
var position : int = middle.IndexOf(30) // 1, rather than 2
```

For `str` elements, `.IndexOf()` compares the text contents, just like `==`. The comparison is case-sensitive and works on fixed arrays, dynamic arrays, and slices of `str`. Separate strings containing the same text match:

```
var names : [*]str = ["Amy", "Lin", "Sam", "Lin"]
var wanted : str = "Lin"
var position : int = names.IndexOf(wanted) // 1: the first matching name
var missing : int = names.IndexOf("lin")  // -1: case matters
```

For other element types, the search compares the bytes stored for each element. This works directly for values such as integers and bytes. Object references match only when they point to the same object; the search does not compare object contents. For structures, floating-point values, or other types that need a custom comparison, use a loop with the comparison you want. Structure padding and different floating-point bit patterns can otherwise affect the result.

2.11.11.2 Equality

For dynamic arrays and array slices, `== []` checks whether the length is zero and `<> []` checks whether there are any elements:

```
var jobs : [*]str
var empty : bool = jobs == [] // true
jobs.Append("download")
var has_jobs : bool = jobs <> [] // true

var pending : []str = jobs[:]
var has_pending : bool = pending <> [] // true
var finished : []str = jobs[1:]
var all_done : bool = finished == [] // true
```

To compare the contents of two arrays, compare their lengths and then their elements in a loop. General array equality is not provided automatically.

2.11.12 Multidimensional Arrays

An array can contain other arrays. Use this to represent a grid or table. For example, `[2][3]int` has two rows, each containing three integers:

```
var matrix : [2][3]int = [ [1, 2, 3], [4, 5, 6] ]
var value : int = matrix[1][2] // 6: second row, third column
matrix[0][1] = 20              // first row is now [1, 20, 3]
```

You can also nest dynamic arrays when rows need to grow separately. Each row has its own storage. Cloning the outer array copies the row references, so the copy still shares those rows; see `.Clone()`.

2.11.13 Dynamic Array `autofree` Elements

Use an element type such as `autofree OTask` when the array should automatically free the objects stored in it. Replacing an element frees its previous object. Removing elements with `.Delete()`, `.Clear()`, or a smaller length or capacity also frees the removed objects. Any remaining objects are freed when the last reference to the dynamic array is released. Assigning the array to another variable still shares the same array, rather than copying its objects.

New `autofree` elements start as `nil`, including those added by `.SetLength()`. Assigning `nil` to an element frees its current object and leaves that position empty, without changing the array's length. A `nil` element has nothing to free.

Assigning an `autofree` variable to an array element transfers the object and sets the source variable to `nil`. Single-element `.Append()`, `.Prepend()`, and `.Insert()` do the same:

```
object OTask:
    name : str
    func *Create(aname : strslice):
        name = aname
    endfunc
endobject

func *Main():
    var tasks : [*]autofree OTask
    tasks.SetLength(2) // [nil, nil]

    var task = new autofree OTask('download')
    tasks[0] = task    // task is now nil; tasks[0] refers to the object
    tasks[0] = nil     // frees the object; tasks is [nil, nil], length still 2

    tasks.Append(new OTask('list'))
    var next : autofree OTask = tasks.Pop() // removes the object without freeing it
endfunc // next is freed automatically
```

`.Pop()` and `.PopFirst()` remove an element without freeing its object. They return an ordinary object reference: assign it to an `autofree` variable, as above, or explicitly delete it when finished. Popping a `nil` element returns `nil`; popping an empty array is still an error. Likewise, check for `nil` before accessing an element's fields or calling its methods.

Copying a range of `autofree` elements and calling `.Clone()` are not supported, because two separate arrays must not automatically free the same objects. See the automatic cleanup rules for more details.

2.12. Enumerations

An enumeration, or **enum**, gives names to a set of choices. Use one for values such as an order's status, a traffic light's color, or a game's difficulty. Each choice is called an **enumerator**.

Enum type names conventionally start with a capital `N`, as in `NOrderStatus`.

Here is a small program that changes an order's status:

```
use print

enum NOrderStatus = (pending, shipped, delivered)

func *Main():
    var status : NOrderStatus = pending
    status = shipped
    if status == shipped:
        PrintLn("Your order is on its way.")
    endif
endfunc
```

This prints `Your order is on its way.`. The enum makes the choices clear and prevents an unrelated number or a value from another enum being assigned to `status`.

2.12.1 Declaration

Write `enum`, a type name, and the choices in parentheses:

```
enum NColor = (red, green, blue)
```

The choices have integer values behind the scenes. By default, the first is zero and each following choice adds one: `red` is `0`, `green` is `1`, and `blue` is `2`. You normally work with the names rather than these numbers.

2.12.1.1 Explicit Values and Storage

In the declaration, assign an integer value with `=` after a choice's name. You can also select the storage type with `: type` after the enum type name:

```
enum NState : uint16 = (
    idle = 0,
    running = 10,
    stopped = 20,
)

enum NSigned : int8 = (negative = -1, zero, positive)
```

Explicit values must be compile-time integer constants. After an explicit value, the next implicit value is one higher: `NSigned.zero` is `0` and `NSigned.positive` is `1`. Values must be unique and increase in declaration order; a later explicit value cannot move backward.

Only declared values belong to the enum. For example, `NState` has values `0`, `10`, and `20`; `11` is not an enum member. Explicit values are useful when matching a file format or an external interface.

The default storage type is `uint8`. You can select `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, or `uint64` to fit the required range. For example, `NState` above occupies two bytes, while `NSigned` occupies one byte and allows negative values. Every choice, including an implicit value, must fit the storage type. The enum remains a separate type from the integer used to store it.

2.12.2 Naming and Lookup

When the expected enum type is clear, use the choice's name directly. You can also include the enum type name:

```
var color : NColor = green
color = NColor.blue
var is_blue : bool = color == blue // true
```

The shorter form works in assignments, function arguments, return values, and comparisons when the compiler can identify one expected enum type. Write `NColor.green` when that context is missing or the name could be ambiguous.

2.12.3 Type Safety

Compare values of the same enum with `==`, `<>`, `<`, `<=`, `>`, or `>=`. Ordering follows their integer values, so `red < blue` when both are used as `NColor` values.

Enums do not support arithmetic or automatic conversion to integers. For example, assigning `1` to an `NColor` variable is an error even though `green` has that integer value. Use the conversion helpers below when reading numbers from a file or another program.

2.12.4 Ordinal Conversion

Use the read-only `.ord` property or `Ord(value)` to get a choice's integer value:

```
var color : NColor = green
var code : uint8 = color.ord // 1
```

To turn a number into an enum, use the enum type's methods. These methods check that the number belongs to a declared choice:

Method	Use it when
`NColor.FromOrd(number)`	The number must be valid; an unknown value raises a runtime error.
`NColor.FromOrd(number, fallback)`	An unknown value should use a fallback choice.
`NColor.TryFromOrd(number, output)`	You want to check success before using the result.

For example, decode a saved color with a fallback:

```
var saved_code : int = 99
var color : NColor = NColor.FromOrd(saved_code, red) // red: 99 is unknown

if NColor.TryFromOrd(2, color):
    // color is now blue
endif
```

`TryFromOrd` returns `true` and updates `output` for a valid number. Otherwise it returns `false` and leaves `output` unchanged. Use this method or a fallback when a file or message might contain choices added by a newer program.

2.13. Structures and Unions

A **structure** keeps related values together. For example, a point has an `x` and a `y` coordinate, and a measurement might have a reading and a timestamp. Each named value inside a structure is a **field**.

Structure type names conventionally start with a capital `S`, as in `SPoint`.

```
use print

struct SPoint:
    x : int
    y : int
endstruct

func *Main():
    var point : SPoint = {x: 10, y: 20}
    var copy : SPoint = point
    copy.x = 30
    PrintLn("{} {}", [point.x, copy.x])
endfunc
```

This prints `10 30`. Assigning a structure copies its fields, so changing `copy.x` leaves `point.x` unchanged.

A **union** lets several fields share the same storage. It is useful for binary formats and C interfaces, and is covered after the structure examples.

2.13.1 Structures

Declare fields between `struct Name:` and `endstruct`. Use a dot to read or change a field:

```
struct SPoint:
    x : int
    y : int
endstruct

var point : SPoint
point.x = 10
point.y = 20
point.y += 5 // point is now {x: 10, y: 25}
var origin : SPoint = {} // both coordinates are zero
```

A structure stores its fields directly in the variable that contains it.

Structures cannot contain managed strings, dynamic arrays, object references, embedded objects, or aggregates with managed fields. Use an object when you need those fields.

For short text stored inside a structure, use a fixed-storage embedded string:

```
struct SProduct:
    name : embstr(32)
    price : float
endstruct

var product : SProduct = {}
product.name = "Bread"
product.price = 2.5
```

Each structure declaration defines a separate type: two structures with the same fields are still different types.

2.13.1.1 Structure Initialization

A local structure variable declared without an initializer starts uninitialized. Reading it immediately produces a `VarNotInit` compiler error. Use `= {}` to explicitly initialize all fields to their zero/default values:

```
func Example():
    var point : SPoint       // uninitialized local variable
    var origin : SPoint = {} // x and y are zero
    point = {10, 20}         // initialize before reading
endfunc
```

Global structure variables and structure fields embedded in objects are zero-initialized by default when no initializer is supplied.

2.13.1.2 Structure Initializer Literals

A brace literal supplies the initial field values. You can name the fields or list values in field declaration order:

```
var named : SPoint = {x: 10, y: 20}
var positional : SPoint = {10, 20}
named = {y: 40, x: 30} // named fields may appear in either order
```

Use a final `?` to give all remaining fields their default values:

```
var point : SPoint = {x: 10, ?} // y is zero
var origin : SPoint = {}       // all fields use their defaults
```

Without `?`, supply every field exactly once. `{?}` also initializes the whole structure to its defaults. The `?` fill marker always have to be the last.

The compiler checks and interprets the structure literals according the expected structure type supplied by a variable or function argument.

Nested initialization literals can be used for structures containing structure fields. Here, `start` and `finish` are both `SPoint` fields, so each inner pair of braces creates an `SPoint`:

```
struct SLine:
    start : SPoint
    finish : SPoint
endstruct

var line : SLine = {start: {0, 0}, finish: {10, 20}}
```

You can initialize a fixed array of structures with a single literal:

```
var points : [2]SPoint = [{0, 0}, {10, 20}]
```

You can mix positional and named structure entries. After a named entry, the next positional entry fills the field following it in declaration order:

```
var point : SPoint = {x: 10, 20} // x is 10; y is 20
```

A structure literal is a temporary value and cannot be passed directly to a reference parameter; put it in a variable first. Structure literals may also initialize compile-time constants, including public constants imported from another module. Union constants are not supported.

2.13.2 Storage Size with `SizeOf()`

`SizeOf(Type)` returns a type's storage size in bytes. `SizeOf(expression)` uses the expression's static type and leaves the expression unevaluated. Both forms produce a compile-time integer constant.

```
const POINT_SIZE : int = SizeOf(SPoint)
var point : SPoint = {}
var bytes : int = SizeOf(point) // same as SizeOf(SPoint)
```

For a structure, the size includes all fields and any padding between or after them. For a union, it is the size of the shared storage, including any padding. Alignment and `[[packed]]` can affect the result; see Structure Field Alignment and Padding.

2.13.3 Field Offsets with `OffsetOf()`

`OffsetOf(Type, field)` returns the number of bytes from the start of a structure to the named field. Pass the type name and the field name without quotes. The result is a compile-time integer constant; no structure variable is needed:

```
struct SHeader [[packed]]:
    kind : uint8
    length : uint32
endstruct

const KIND_OFFSET : int = OffsetOf(SHeader, kind)     // 0
const LENGTH_OFFSET : int = OffsetOf(SHeader, length) // 1
```

Here, `length` starts one byte after the start of the structure. `OffsetOf()` includes any padding before a field, so removing `[[packed]]` may change the result. See Structure Field Alignment and Padding for alignment and padding rules.

Use `OffsetOf()` when checking a binary layout or matching an external record. `OffsetOf()` also accepts union types, where every field has offset zero.

2.13.4 Structure Methods

A method is a function that belongs to a type. Use one to perform an operation on the structure's fields:

```
struct SSize:
    width : int
    height : int

    func Area() -> int:
        return width * height
    endfunc
endstruct

var size : SSize = {width: 4, height: 3}
var area : int = size.Area() // 12
```

Inside the method, fields are available by name. Methods can also be declared inside the structure and defined later with a qualified name, such as `func SSize.Area() -> int:`.

Structure methods do not have the virtual dispatch described for objects.

2.13.5 Structure Inheritance

A structure can inherit fields and methods from one base structure:

```

struct SPoint:
    x : int
    y : int
endstruct

struct SPoint3D(SPoint):
    z : int
endstruct

var point : SPoint3D = {x: 10, y: 20, z: 30}
```

When inintializing a derived structure base fields come first, followed by derived fields.

2.13.6 Storage Unions

A union provides several ways to interpret the same bytes. Declare it with `union Name:` and `endunion`:

```
union UWord:
    raw : uint32
    bytes : [4]byte
endunion

var word : UWord = {raw: 42}
var number : uint32 = word.raw // 42
```

All fields begin at the same address. Writing `word.raw` also changes the bytes seen through `word.bytes`. Which byte comes first depends on the target's byte order.

Initialize a union with exactly one field, or use `{}` to zero all its storage. Storage beyond the chosen field receives default initialization. Assignment copies the complete union storage.

A union must contain at least one field. Unions do not support inheritance, methods, properties, field initializers, automatic conversion to a field's type, or union-valued constants. Each union declaration defines a distinct type.

2.13.6.1 Nested Union Fields

A structure or object can contain a named union field. For example, a measurement may store an integer reading or a floating-point reading:

```
struct SReading:
    kind : int
    union data:
        integer : int32
        real : float32
    endunion
endstruct

var reading : SReading = {kind: 1, data.integer: 42}
var value : int32 = reading.data.integer // 42
```

Here, the program can use `kind` to remember which form it stored. DQ does not maintain this field automatically. `data.integer: 42` is a shorter spelling of `data: {integer: 42}`. Select only one alternative for each union field in an initializer; alternatives may themselves contain nested structures or unions.

2.13.6.2 Unnamed Union Fields

Omit the field name to expose the alternatives directly on the containing structure or object:

```
struct UART_REG:
    union:
        RBR : [[regro]] uint32
        DLL : [[regrw]] uint32
        THR : [[regwo]] uint32
    endunion
    union:
        DLH : [[regrw]] uint32
        IER : [[regrw]] uint32
    endunion
    LCR : [[regrw]] uint32
endstruct

var uart : ^UART_REG = ^UART_REG(0x40000000)
uart.DLL = 1
var received : uint32 = uart.RBR
```

`RBR`, `DLL`, and `THR` share offset zero; `DLH` and `IER` share offset four; `LCR` starts at offset eight. Each unnamed union occupies one field's storage, with the same size and alignment rules as a named union field. `OffsetOf()` accepts the exposed names, for example `OffsetOf(UART_REG, IER)` is `4`. The brace spelling is `union { ... }`.

Named initializers also use the exposed names and may select only one alternative from each unnamed union:

```
struct SValue:
    tag : int
    union:
        integer : int32
        real : float32
    endunion
endstruct

var value : SValue = {tag: 1, integer: 42}
var positional : SValue = {1, {integer: 42}}
```

In a positional initializer, the unnamed union takes one entry containing a union initializer. `{}` zeroes all storage as usual.

Exposed names must be unique among the containing type's fields, methods, properties, other unnamed unions, and inherited members. Inside object or structure methods, the exposed fields can be used without `self.`. An unnamed union in an object follows the current `public`, `protected`, or `private` section, and register access attributes on each alternative remain in effect.

2.13.7 Structure Field Alignment and Padding

Fields are stored in declaration order. Each field normally starts at an offset that satisfies its type's **alignment requirement**. For example, an `uint32` field starts at an offset divisible by four. The compiler inserts unused bytes, called **padding**, between fields when needed.

A structure's alignment is the largest alignment required by any of its fields or its base structure. For example, if that alignment is four bytes, the structure must start at an address divisible by four.

Its size must also be a multiple of four in that example. If the fields and the padding between them take 10 bytes, the compiler adds 2 unused bytes at the end, making the size 12 bytes. In an array, the next structure then starts 12 bytes later, again at an address divisible by four.

2.13.7.1 Packed Structures

Put `[[packed]]` after the structure name, before the colon, to store fields consecutively without adding padding between them or after the last field. A packed structure has one-byte alignment:

```
struct SRecord:
    tag : byte
    count : uint32
    code : uint16
endstruct

struct SPackedRecord [[packed]]:
    tag : byte
    count : uint32
    code : uint16
endstruct
```

On a target where `uint32` has four-byte alignment and `uint16` has two-byte alignment, these declarations have the following layouts:

Query	Normal structure	Packed structure
`OffsetOf(Type, tag)`	0	0
`OffsetOf(Type, count)`	4	1
`OffsetOf(Type, code)`	8	5
`SizeOf(Type)`	12	7

Packing an outer structure does not change the internal layout of a nested structure field. That field retains its own size and any internal padding. Packed fields may have unaligned addresses; this matters when passing their addresses to external code or hardware.

2.13.7.2 Explicit Field Alignment

Use `[[align(n)]]` after a field declaration to request a larger alignment in bytes:

```
struct SAlignedRecord:
    tag : byte
    value : uint16 [[align(8)]]
endstruct
```

Here, `value` starts at offset `8`, the structure has eight-byte alignment, and its size is `16` bytes. A smaller alignment request does not reduce a field's natural alignment. In a packed structure, field alignment attributes are ignored.

A union uses the strictest alignment required by any field and is large enough for its largest field, rounded up to that alignment. `[[packed]]` is not supported on unions.

Use fixed-width fields and check `SizeOf` and `OffsetOf` when matching a C record or hardware interface. These queries, member access, and address-taking also work with unions. See Attributes and C Interoperability.

2.14. Pointers

A **pointer** stores a memory address, which identifies a location in memory.

2.14.1 Pointer Basics

For a data type `T`, write `^T` to create the corresponding pointer type, such as `^int` or `^float`.

When working with pointers, it is important to distinguish between **address operations** and **pointed-data operations**.

Address operations work on the memory address stored in the pointer. DQ works on the address when **neither `^` nor `.`** is written **after the name** of the pointer variable or constant:
`ptr = address`

Pointed-data operations work on the data at the address stored in the pointer. DQ works on that data when **`^` or `.`** is written **after the name** of the pointer variable or constant. This is also called **dereferencing**:
`ptr^ = value`

A simple example using the `%` to get the address of a variable:

```
var u32value : uint32       // an ordinary variable whose storage is managed by the compiler
u32value = 0x04030201       // assigns a value to the variable

var byteptr : ^byte         // declares a pointer variable of type `^byte`
byteptr = ^byte(%u32value)  // points to the memory location where `u32value` is stored
byteptr^ = 0x55             // updates the first byte at that address

PrintLn('u32value = 0x{:08X}', [u32value])  // = "u32value = 0x04030255"
```

On a little-endian system, the first byte is the least significant byte of `u32value`, so the example prints the result shown above.

Pointers are widely used and can **improve performance** significantly.

Using pointers also **can be dangerous**. There are many ways to end up with an invalid pointer (one that points to an invalid memory location). Accessing data through an invalid pointer can return invalid data or cause the application to crash.

Pointers are useful when working with C libraries, raw buffers, or hardware. For an ordinary function that updates a caller's variable, a `ref` parameter is usually simpler.

For a more practical example for using pointers see the Example: Processing a Sensor Packet.

2.14.2 Pointer Variables

Assigning a pointer variable sets its pointing address. A pointer variable can be re-assigned to point a different address:

```
var first  : int = 10
var second : int = 20
var curptr : ^int

curptr = %first    // setting the address of the pointer
curptr^ = 15       // first is now 15; current^ also reads 15
curptr = %second   // redirects curptr to second
curptr^ = 25       // second is now 25; first remains 15

var otherptr : ^int = current // assignment between two pointer variables
```

2.14.3 Typed Pointers and Casting

A pointer's type tells the compiler how to access the data at its address: how many bytes to read or write, which structure fields are available, and how far to move during pointer arithmetic.

The type after `^` describes the pointed-to value. For example, `^byte` accesses a single byte, `^int32` accesses a 4-byte signed integer, and `^float` accesses a floating-point number occupying 4 or 8 bytes, depending on the target. (The size of the pointer itself is the same for all three types, either 4 or 8 bytes depending on the target CPU architecture.)

The pointed-to type also determines the step size of pointer arithmetic and indexing: adding 1 to a `^byte` moves by 1 byte, while adding 1 to a `^uint32` moves by 4 bytes.

Casting converts a value from one type to another, or tells the compiler to treat a value as a different type.

For pointers, casting usually changes the **pointer type** without changing the address stored in the pointer. The same memory address is then interpreted as pointing to a different data type.

The cast tells the compiler how to interpret the address; it does not check the actual value stored there.

Sometimes an address comes from an API without type information, or the same memory needs to be accessed through a different pointer type. An explicit cast supplies the required type.

Write `^T(address)` to cast an address to a pointer to `T`. A pointer cast does not read, copy, or convert the data stored there.

For example, `^byte`, `^int32`, `^float`, and `^uint64` pointers can all point to the beginning of the same eight-byte buffer. Use `%` to take its address and a trailing `^` to read or write through each pointer:

```
use print

var buffer : [8]byte [[align(8)]]

func ShowBuffer(acaption : strslice):
    Print("{:<28} at {:p}:", [acaption, %buffer[0]])
    for i : int = 0 count buffer.length:
        Print(" {:02X}", [buffer[i]])
    endfor
    PrintLn()
endfunc

func *Main():
    var byteptr  : ^byte   = %buffer[0]
    var intptr   : ^int32  = ^int32(%buffer[0])
    var floatptr : ^float  = ^float(%buffer[0])
    var u64ptr   : ^uint64 = ^uint64(%buffer[0])

    ShowBuffer("Initial")

    for i : int = 0 count buffer.length:
        byteptr[i]^ = 0xD0 + i
    endfor
    ShowBuffer("Set Pattern")

    byteptr^ = 42    // writes 1 byte
    ShowBuffer("byteptr^ = 42")

    intptr^ = -1000  // writes 4 bytes
    ShowBuffer("intptr^ = -1000")

    floatptr^ = 1.5  // writes SizeOf(float) bytes
    ShowBuffer("floatptr^ = 1.5")

    u64ptr^ = 0x0102030405060708 // writes 8 bytes
    ShowBuffer("u64ptr^ = 0x0102030405060708")
endfunc
```

Example output on Linux x64 (the address varies between runs):

```
Initial                     at ^555512345678: 00 00 00 00 00 00 00 00
Set Pattern                 at ^555512345678: D0 D1 D2 D3 D4 D5 D6 D7
byteptr^ = 42               at ^555512345678: 2A D1 D2 D3 D4 D5 D6 D7
intptr^ = -1000             at ^555512345678: 18 FC FF FF D4 D5 D6 D7
floatptr^ = 1.5             at ^555512345678: 00 00 00 00 00 00 F8 3F
u64ptr^ = 0x0102030405060708 at ^555512345678: 08 07 06 05 04 03 02 01
```

`ShowBuffer` prints the starting address followed by all eight bytes in memory order. The global buffer starts with zeroes.

The "Set Pattern" loop writes the pattern using pointer indexing through `byteptr[i]^`, without changing `byteptr`.

The following `byteptr^ = 42` changes only the first byte of that pattern. The `int32` write changes the first 4 bytes to the two's-complement representation of `-1000`, leaving `D4 D5 D6 D7` unchanged. Almost all current CPU architectures use little-endian byte order, with the least significant byte first. The floating-point write uses 8 bytes and shows the `float64` representation of `1.5`. On targets with a 4-byte `float`, that write changes only the first 4 bytes. The final `0x0102030405060708` `uint64` write always uses 8 bytes: `08 07 06 05 04 03 02 01` in little-endian byte order.

The global buffer has eight-byte alignment, requested by `[[align(8)]]`, so it can hold any of these values at its starting address, as some CPU architectures cannot read/write multi-byte data types from unaligned addresses.

`%buffer[0]` takes the address of its first byte. The casts `^int32(%buffer[0])`, `^float(%buffer[0])`, and `^uint64(%buffer[0])` take the same address and give it different pointer types.

As all four pointers access the same storage, each write overwrites the bytes used by that type, and each read interprets those bytes according to the pointer's type. Reading through `intptr^` after the floating-point write would interpret the low 4 bytes of the floating-point representation as an integer (=0); it would not convert `1.5` to an integer.

2.14.4 The `nil` Value

`nil` is the special pointer value meaning **no target**. It is also called a **null-pointer** and represents address zero. Both typed pointers, such as `^int`, and generic `pointer` variables can be assigned `nil`.

A `nil` pointer is **always invalid for accessing data**. Dereferencing it to read or write data, either with `ptr^` or through a structure field such as `ptr.field`, usually causes the program to crash with a **Segmentation Fault**. The exact failure depends on the target system; never dereference `nil`.

Assigning, copying, and comparing `nil` are safe address operations. Use `==` or `<>` to check whether a pointer has a target before accessing its data:

```
var value : int = 10
var ptr   : ^int = nil // no target yet

if ptr == nil:
    ptr = %value      // give the pointer a valid target
endif

if ptr <> nil:
    ptr^ += 1         // value is now 11
endif

ptr = nil             // clear the pointer; value remains 11
```

Setting a pointer to `nil` clears its stored address; it does not change or free the pointed-to data, unless it is an `autofree` pointer.

2.14.5 Pointers as Function Parameters

Typed pointers can also be function parameters. A pointer may be `nil`, meaning it has no target, so a function that accepts an optional target must check before dereferencing it:

```
func Increment(target : ^int):
    if target <> nil:
        target^ += 1
    endif
endfunc

func *Main():
    var count : int = 10
    Increment(%count) // count is now 11
    Increment(nil)    // no target, so nothing is changed
endfunc
```

A non-`nil` pointer must still refer to a live value of the correct type. Copying a pointer does not copy its target or keep that target alive. DQ does not generally check raw addresses for you.

2.14.6 Structure Pointers

Pointers tor structures are very useful to manipulate memory structures at arbitrary positions in the memory. Fields of the pointed structure can be accessed either with `^.` or `.` only.

```
struct SPoint:
    x : int
    y : int
endstruct

var point   : SPoint = {x: 10, y: 20}
var address : ^SPoint = %point
address^.x = 30 // changes point.x
address.x  = 30 // changes point.x, `.` dereferences automatically
```

2.14.7 Generic Pointers

A generic `pointer` carries no information about the data it points to. Generic pointers do not support typed element arithmetic until converted to a suitable typed pointer.

Cast it to a typed pointer before accessing that data:

```
var value  : uint32 = 0x01020304
var rawptr : pointer = %value

^byte(rawpr)^ = 0x55 // value is now 0x01020355
```

Here, `^byte(rawptr)` produces a pointer of type `^byte`, so the `^byte(rawpr)^` expression manipulates the lowest byte of the 4-byte `value`.

2.14.8 Pointer Size and Memory Addresses

A pointer variable is either **32 bits (4 bytes)** or **64 bits (8 bytes)** wide, depending on the target CPU architecture. This is the size of the stored address; the data it points to can have a different size. For example, `^byte` and `^uint64` have the same size on a given target.

In DQ, `int` and `uint` always have the same size as a pointer: `SizeOf(pointer)`, `SizeOf(int)`, and `SizeOf(uint)` are equal.

Target	Pointer size	`int`	`uint`
ARM_M (Arm Cortex-M)	32 bits (4 bytes)	`int32`	`uint32`
Linux x64 (`x86_64-linux`)	64 bits (8 bytes)	`int64`	`uint64`

Memory addresses are usually written in **hexadecimal**, where each digit represents 4 bits. A full-width 32-bit address has 8 hexadecimal digits, while a full-width 64-bit address has 16. Leading zeroes are often omitted.

For example, addresses might look like this:

Target	Example address	Meaning
ARM_M	`0x20001234`	A location in the SRAM region
ARM_M	`0x40001000`	A location in the peripheral region
Linux x64	`0x0000555512345678`	A location in a process's virtual address space
Linux x64	`0x00007FFC1234ABCD`	Another possible user-space virtual address

On ARM_M, addresses refer directly to memory or hardware registers. The Cortex-M memory map defines regions such as SRAM and peripherals; the exact locations available depend on the microcontroller.

On Linux x64, an ordinary program uses **virtual addresses**, which the operating system maps to physical memory. Addresses can vary between runs. A pointer still occupies 8 bytes even when its address uses fewer than 16 hexadecimal digits; the CPU and operating system do not make every possible 64-bit address usable. You can see the Linux x64 memory map for more details.

DQ's `PrintLn` displays pointer addresses in hexadecimal with a leading `^` instead of `0x`:

```
var value   : int = 42
var address : ^int = %value
PrintLn("{:p}", [address])
```

Example output could be `^20001234` on ARM_M or `^7FFC1234ABCD` on Linux x64. These are illustrative addresses; the actual address depends on where `value` is stored.

2.14.9 Pointer Arithmetic and Indexing

Pointer arithmetic counts **elements of the pointed-to type**. For a pointer `ptr : ^T`, adding `n` moves the address by `n * SizeOf(T)` bytes.

Operation	Meaning	Changes `ptr`?
`ptr + n`	Address `n` elements forward	No
`ptr - n`	Address `n` elements backward	No
`++ptr` / `--ptr`	Move forward / backward by one element	Yes
`ptr += n` / `ptr -= n`	Move forward / backward by `n` elements	Yes
`ptr[n]`	Same address as `ptr + n`	No

Pointer indexing returns a **pointer**, so `ptr[n]^` reads or writes the selected value. For a structure pointer, use `ptr[n]^.x` to access a field of the selected structure.

2.14.9.1 Stepping Through the Same Buffer

This example uses a fixed `[16]byte` buffer. It first advances a byte pointer, then casts its current address to `^uint32` and continues in 4-byte steps. The structure writes start at byte offset 8. Here, `SPoint` uses `int16` coordinates so that each point occupies 4 bytes on both 32-bit and 64-bit targets.

Pointer type	Element size	Bytes moved by one `++`	Byte offset selected by `[1]` from the start
`^byte`	1 byte	1	1
`^uint32`	4 bytes	4	4
`^SPoint`	4 bytes	4	4

The same 16 bytes can therefore hold 16 bytes, four `uint32` values, or four `SPoint` values. These are different views of the same storage; writes through any of the pointers change `buffer`.

The `uint32` writes below use unaligned addresses, so run this example on a target that supports them, such as Linux x64.

```
use print

struct SPoint:
    x : int16
    y : int16
endstruct

var buffer : [16]byte [[align(8)]]

func ShowBuffer(caption : strslice):
    Print("{:<18}:", [caption])
    for i : int = 0 count buffer.length:
        Print(" {:02X}", [buffer[i]])
    endfor
    PrintLn()
endfunc

func *Main():
    var byteptr  : ^byte   = %buffer[0]
    for i : int = 0 count buffer.length:
        byteptr[i]^ = i // indexing leaves byteptr at the start
    endfor
    ShowBuffer("Initial")

    byteptr^ = 0xAA // writes 1 byte at offset 0
    ++byteptr       // moves 1 byte: now at buffer[1]
    byteptr^ = 0xBB // writes 1 byte at offset 1
    ++byteptr       // moves 1 byte: now at buffer[2]
    ShowBuffer("After ++byteptr")

    var u32ptr : ^uint32 = ^uint32(byteptr) // keeps byte offset 2
    u32ptr^ = 0xD1D2D3D4 // writes 4 bytes at offsets 2..5
    ++u32ptr             // moves 4 bytes: now at buffer[6]
    u32ptr^ = 0xE1E2E3E4 // writes 4 bytes at offsets 6..9
    ++u32ptr             // moves 4 bytes: now at buffer[10]
```

```
    ShowBuffer("After ++u32ptr")

    // Start at offset 8 so that two 4-byte points fit in the buffer.
    var pointptr :? = ^SPoint(%buffer[8])
    pointptr.x = 11 // writes 2 bytes at offsets 8..9
    pointptr.y = 21 // writes 2 bytes at offsets 10..11
    ++pointptr      // moves 4 bytes: now at buffer[12]
    pointptr.x = 12 // writes 2 bytes at offsets 12..13
    pointptr.y = 22 // writes 2 bytes at offsets 14..15
    ShowBuffer("After ++pointptr")

    // Reset all three pointers to the beginning before comparing indexing.
    byteptr  = %buffer[0]
    u32ptr   = ^uint32(%buffer[0])
    pointptr = ^SPoint(%buffer[0])

    byteptr[2]^ = 0xCC       // byte offset 2
    u32ptr[1]^ = 0xA1A2A3A4  // byte offset 4: the second uint32
    pointptr[2]^.x = 0x5353  // writes 2 bytes at offsets 4..5: the second point's x
    pointptr[2]^.y = 0x5252  // writes 2 bytes at offsets 6..7: the second point's y
    ShowBuffer("Indexed writes")

    PrintLn("Second point: ({}, {})", [pointptr[1]^.x, pointptr[1]^.y])

    var fourth : ^SPoint = pointptr + 3 // byte offset 12: the fourth point
    PrintLn("Distance: {} points, {} bytes",
        [fourth - pointptr, ^byte(fourth) - ^byte(pointptr)])
endfunc
```

Example output on Linux x64 (little-endian):

```
Initial           : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
After ++byteptr   : AA BB 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
After ++u32ptr    : AA BB D4 D3 D2 D1 E4 E3 E2 E1 0A 0B 0C 0D 0E 0F
After ++pointptr  : AA BB D4 D3 D2 D1 E4 E3 0B 00 15 00 0C 00 16 00
Indexed writes    : AA BB CC D3 A4 A3 A2 A1 53 53 52 52 0C 00 16 00
Second point: (41892, 41378)
Distance: 3 points, 12 bytes
```

Each `++` changes only the pointer's address; the assignments through the pointers change the buffer. Casting `byteptr` to `^uint32` keeps its current byte offset, 2; the subsequent `++u32ptr` operations step to offsets 6 and 10. The structure writes at offsets 8..15 overwrite the last two bytes of the previous `uint32` write as well as the remaining initial bytes.

After the reset, the `[]` expressions leave all three pointers at the beginning. For example, `u32ptr[1]^` writes the same four bytes as `(u32ptr + 1)^`. `pointptr[1]^.x` selects the second point, dereferences it, and accesses its `x` field. The indexed `SPoint` writes replace the same four bytes at offsets 4..7 that were just written through `u32ptr[1]^`.

Subtracting two pointers of the same type gives their distance in **elements**. The two point addresses above are three `SPoint` elements apart. Casting both to `^byte` before subtracting gives the distance in bytes: 12.

The buffer starts at an address aligned with `[[align(8)]]`, but offsets 2 and 6 are still unaligned for `uint32`. On targets that require aligned accesses, start those writes at offsets divisible by 4. A cast alone does not provide alignment or make more storage available. Unlike array indexing, pointer indexing carries no length information and does not check bounds. Keep each entire value you read or write within the buffer. You may form a pointer just past the last element, but must not dereference it; subtract pointers only within the same allocation or its one-past-end position.

2.14.10 Allocation and Lifetime

Use `new` when the pointed-to value needs separately allocated storage, and initialize it before reading it. Use `delete` when finished:

```
struct SPoint:
    x : int
    y : int
endstruct

var point : ^SPoint = new SPoint
point^ = {x: 10, y: 20}
delete point = nil // release the allocation and clear this variable
```

Other pointers to the same allocation become invalid after `delete`; they are not cleared automatically. To release an allocation automatically when leaving a scope, use `autofree ^T`. See Memory Management and Autofree.

Pointers into local variables or fixed arrays are valid only while that storage exists. Resizing a dynamic array or growing or changing a string's storage can also invalidate pointers into it. Obtain a fresh pointer after such a change. Never return a pointer to a function's local variable.

2.14.11 Dangling Pointers

A non-`nil` address does not always guarantee that a pointer is valid: it might refer to memory that has already been freed or to a variable that has gone out of scope, like a local variable. Such pointers are called **dangling pointers**.

Using `SPoint` from the allocation example above:

```
var point : ^SPoint = new SPoint
point^ = {x: 10, y: 20}
var borrowed : ^SPoint = point // both pointers refer to the same allocation

delete point = nil          // free the allocation and clear point
// borrowed is still non-nil, but is now dangling:
// borrowed.x = 30          // invalid: the target has already been freed
borrowed = nil              // clear the dangling pointer
```

Never dereference `nil` or dangling pointers.

2.14.12 Pointer Examples

2.14.12.1 A Linked List with First and Last Pointers

A **linked list** stores each value in a node with a pointer to the next node. The nodes can be allocated separately. Unlike an array, the list is traversed by following `next`, rather than by incrementing a pointer.

`SList` keeps two addresses: `first` selects the first node and `last` selects the final node. Both are `nil` for an empty list. Keeping `last` lets `Append` add a node without walking through all the existing nodes. The last node's `next` is always `nil`.

```
use print

struct SNode:
    value : int
    next  : ^SNode
endstruct

struct SList:
    first : ^SNode
    last  : ^SNode

    func Append(value : int):
        var node : ^SNode = new SNode
        node.value = value
        node.next = nil

        if last == nil:
            first = node // adding the first node to an empty list
        else:
            last.next = node
        endif
        last = node
    endfunc

    func PopFirst(value : ref int) -> bool:
        if first == nil:
            return false
        endif

        var node : ^SNode = first
        value = node.value
        first = node.next // save the next address before releasing the node
        if first == nil:
            last = nil // removing the only remaining node empties both ends
        endif
        delete node
        return true
    endfunc

    func Clear():
        var value : int = 0
        while PopFirst(value):
        endwhile
    endfunc
endstruct

func *Main():
    var list : SList = {} // both first and last start as nil
```

```
    list.Append(10)
    list.Append(20)
    list.Append(30)
    PrintLn("First: {}, last: {}", [list.first.value, list.last.value])

    Print("Values:")
    var current : ^SNode = list.first
    while current <> nil:
        Print(" {}", [current.value])
        current = current.next // follow the link; nodes need not be adjacent
    endwhile
    PrintLn()

    var value : int = 0
    while list.PopFirst(value):
        PrintLn("Removed: {}", [value])
    endwhile
    PrintLn("Empty: {}", [list.first == nil and list.last == nil])

    list.Append(40) // the emptied list can be used again
    PrintLn("Reused: first={}, last={}", [list.first.value, list.last.value])
    list.Clear() // release every node still owned by the list
    PrintLn("Cleared: {}", [list.first == nil and list.last == nil])
endfunc
```

Example output:

```
First: 10, last: 30
Values: 10 20 30
Removed: 10
Removed: 20
Removed: 30
Empty: true
Reused: first=40, last=40
Cleared: true
```

When appending to an empty list, the new node becomes both `first` and `last`. Otherwise, `last.next` links the old tail to the new node, and `last` moves to that node. Appending changes links, without moving any existing nodes.

`PopFirst` returns `false` for an empty list. Otherwise, it copies the first node's value into the caller's variable, advances `first` to `node.next`, and then deletes the removed node. It clears `last` when the last remaining node is removed, so a later `Append` correctly starts a new list. Both appending and removing the first node take a constant number of pointer operations.

The list owns its allocated nodes. `Clear` reuses `PopFirst` to release all of them; deleting one node does not automatically follow and delete its `next`. Call `Clear` before the list leaves scope if it still contains nodes. Copying an `SList` structure copies only its two addresses, so it does not create an independent list and must not create a second owner of the same nodes. Any borrowed pointer to a removed node becomes invalid when that node is deleted.

2.14.12.2 Processing a Sensor Packet

This is a longer, somewhat practical example demonstrating using pointers.

Suppose a sensor station sends a binary packet containing a header followed by measurement records. The header tells us the format version, measurement time, location, and number of records. Each record identifies a measurement type and channel, and stores a measured value. In this example, measurement type `1` means temperature in degrees Celsius and type `2` means relative humidity in percent.

A file or network API may supply this packet as a buffer address and a byte count. Pointers let us read the header and records directly in that buffer, without copying each record into a separate variable.

For safer data handling structures are used in practice. Here two structures are used to group the data fields: `SDataHeader` and `SDataRec`. The `SDataHeader` describe the 32-byte header and the `SDataRec` the 8-byte data records.

```
Byte offset:  0                 32             40              48
              | SDataHeader      | SDataRec      | SDataRec      |
              | version, count,  | type, channel,| type, channel,|
              | time, location,  | pad, value    | pad, value    |
              | reserved space   |               |               |
```

The sender must use the same field layout, byte order as the receiver.

The `_pad` fields in this example align the data fields manually and/or reserve space for future expansions. The `_pad` fields initialized with zeroes.

Here, `Main` builds a packet in memory with that layout so the program can run without a file or network connection. `SExamplePacket` only supplies the sample storage; `PrintData` uses the address and byte count and can handle any number of records that fits.

```
use print
use time

struct SDataHeader:
    version  : uint32     // format identifier
    reccount : uint32     // data record count
    mtime    : float64    // measurement time: @time.Time()
    location : uint32     // measurement location
    _pad     : [3]uint32  // reserved bytes for future expansion
endstruct

struct SDataRec:
    mtype      : byte  // measurement type: 1 = temperature, 2 = humidity
    channel    : byte  // one station can have multiple sensors
    _pad       : [2]byte
    value      : float32  // measurement value
endstruct

func PrintData(data : pointer, size : int):
    if data == nil or size < SizeOf(SDataHeader):
        PrintLn("Incomplete header")
        return
    endif

    var header :? = ^SDataHeader(data)
    if header.version <> 1:
        PrintLn("Unsupported data version")
        return
    endif

    var capacity : int = (size - SizeOf(SDataHeader)) div SizeOf(SDataRec)
    if header.reccount > capacity:
        PrintLn("Incomplete records")
        return
    endif

    PrintLn("Location {}, time {}, record count: {}",
        [header.location, FormatTime1('yyyy-mm-dd hh:nn:ss', header.mtime), header.reccount])

    // header[1] points just past the header, where the records begin.
    var record :? = ^SDataRec(header[1])
    for i : uint32 = 0 count header.reccount:
        PrintLn("  mtype={}, channel={}, value={:.1f}",
            [record.mtype, record.channel, record.value])
        ++record
    endfor
endfunc

// Supplies an aligned buffer containing a header and two records.
struct SExamplePacket:
    header  : SDataHeader
    records : [2]SDataRec
endstruct

func *Main():
    var packet : SExamplePacket = {
        header: {version: 1, reccount: 2, location: 7, ?},
        records: [
            {mtype: 1, channel: 0, value: 21.5, ?},
            {mtype: 2, channel: 0, value: 45, ?}
        ]
    }
    // The structure literal can contain constants only. This is not a constant:
    packet.header.mtime = Time()

    PrintData(%packet, SizeOf(packet))
endfunc
```

Example output (the time varies because `Time()` supplies the current time):

```
Location 7, time 2026-10-31 19:30:00, record count: 2
  mtype=1, channel=0, value=21.5
  mtype=2, channel=0, value=45.0
```

In `var header :? = ^SDataHeader(data)`, the cast converts the generic buffer address to `^SDataHeader`, and `:?` tells the compiler to infer `header`'s type from that expression. This allows `header.version` and `header.reccount` to access the header's fields. The cast does not read or copy the whole header.

To find the records, `header[1]` selects the address one **header** past `header`, equivalent to `header + 1`. Because `header` has type `^SDataHeader`, this moves by `SizeOf(SDataHeader)`, or 32 bytes. Pointer indexing returns a pointer, so `header[1]` does not read a second header; it only computes the address where the records begin.

Similarly, `var record :? = ^SDataRec(header[1])` converts that address to `^SDataRec` and infers that pointer type for `record`, letting us read a record's fields. Then `++record` advances by one **record**, or 8 bytes. Both operations use the pointed-to type's size: `header[1]` skips a header, while `record[1]` would point to the next record.

Before accessing records, the function divides the remaining byte count by `SizeOf(SDataRec)` to find how many complete records fit. It checks the announced count against that capacity. Dividing the available space avoids multiplying an unchecked record count, which could overflow. Try passing `SizeOf(packet) - 1` to `PrintData`: it prints `Incomplete records` without reading either record.

The caller must still supply a valid buffer of at least `size` bytes; the function cannot verify that an arbitrary address refers to that much memory. The pointers borrow `packet`'s storage and are used only during the call, while `packet` is alive. No allocation or `delete` is needed. A binary format with a different byte order or floating-point representation needs explicit decoding rather than these direct pointer casts.

2.14.12.3 Double Pointers and an argv-Style Dynamic List

A **double pointer**, such as `^^char`, points to a variable or array element that itself holds a `^char`. This is useful for a list of pointers, such as a C-style argument table (`argv`): each entry points to a zero-terminated string. The strings can have different lengths and do not need to be adjacent in memory.

This example builds the strings in a dynamic array, then builds a separate dynamic array of pointers to their text. It passes the address of the first pointer slot as `^^char`, together with an argument count (`argc`). A final `nil` entry also allows traversal without using the count.

```
use print

func PrintArgs(argc : int, argv : ^^char):
    for i : int = 0 count argc:
        var text : rostr = argv[i]^ // read a ^char from the pointer table
        PrintLn("argv[{}] = {}", [i, text])
    endfor

    var cursor : ^^char = argv
    Print("First characters:")
    while cursor^ <> nil:
        var textptr : ^char = cursor^
        Print(" {}", [textptr^])
        ++cursor // advance to the next ^char slot, not the next character
    endwhile
    PrintLn()
endfunc

func *Main():
    var arguments : [*]str = ["demo", "--color"]
    arguments.Append("blue")

    var pointers : [*]^char
    for i : int = 0 count arguments.length:
        pointers.Append(arguments[i].pchar)
    endfor
    pointers.Append(nil) // the final slot marks the end of the list

    var argv : ^^char = %pointers[0]
    PrintArgs(arguments.length, argv)

    var replacement : str = "green"
    argv[2]^ = replacement.pchar // replace a pointer in the table
    var selected : rostr = pointers[2]
    PrintLn("After replacement: {}", [selected])
    PrintLn("Original string: {}", [arguments[2]])
endfunc
```

Example output:

```
argv[0] = demo
argv[1] = --color
argv[2] = blue
First characters: d - b
After replacement: green
Original string: blue
```

There are two levels of storage between `argv` and a character:

Expression	Type	Meaning
`argv`	`^^char`	Address of the first pointer slot
`argv[i]`	`^^char`	Address of pointer slot `i`
`argv[i]^`	`^char`	String address stored in slot `i`
`textptr^`	`char`	One character at the selected string address

`++cursor` moves by `SizeOf(^char)`: one pointer slot, or 4 or 8 bytes depending on the target. It does not move through the string's characters. To do that, increment the inner `^char` pointer instead.

Assigning `argv[2]^` replaces the pointer stored in `pointers[2]`. It does not copy text, modify the string `"blue"`, or replace `arguments[2]`. The `rostr` variables borrow the selected zero-terminated text so `PrintLn` can print it as a string rather than as an address.

The dynamic arrays manage their own storage. The pointer table borrows the strings; it does not own them. Keep `arguments` and `replacement` alive while their text pointers are used, and obtain fresh `.pchar` values after changing the source strings. Finish building `pointers` before taking `%pointers[0]`: resizing it can invalidate `argv` and `cursor`. The function expects `argc` valid string pointers followed by a `nil` slot; a raw double pointer does not carry the table's length.

2.14.12.4 Using Double Pointers for Return Values

A function can return a value normally and supply another result through a pointer parameter. To return a `^SList` this way, the parameter must be `^^SList`: the address of the caller's pointer variable.

For example, `CollectHighReadings` builds a list of temperature readings at or above a threshold. It returns the number of matching readings and writes the new list's address through its last parameter, `outlist`.

Reuse `SNode` and `SList` from A Linked List with First and Last Pointers, and replace that example's `Main` with the following code:

```
func CollectHighReadings(readings : []int, threshold : int, outlist : ^^SList) -> int:
    if outlist == nil or outlist^ <> nil:
        return -1 // require an output pointer whose current value is nil
    endif

    var list : ^SList = new SList
    list^ = {} // initialize first and last to nil
    var count : int = 0
    for reading : int in readings:
        if reading >= threshold:
            list.Append(reading)
            ++count
        endif
    endfor

    outlist^ = list // update the caller's pointer variable
    return count
endfunc

func *Main():
    var readings : [?]int = [22, 31, 24, 35]
    var alerts : ^SList = nil
    var count : int = CollectHighReadings(readings, 30, %alerts)
    if count < 0:
        PrintLn("Invalid output pointer")
        return
    endif

    PrintLn("High readings: {}", [count])
    var current : ^SNode = alerts.first
    while current <> nil:
        PrintLn("Reading: {}", [current.value])
        current = current.next
    endwhile

    alerts.Clear()      // release the list's nodes
    delete alerts = nil // release the SList itself and clear the pointer
endfunc
```

Example output:

```
High readings: 2
Reading: 31
Reading: 35
```

`alerts` has type `^SList`, so `%alerts` has type `^^SList`. Inside the function, `outlist^` accesses the caller's `alerts` variable. Assigning `outlist^ = list` therefore changes `alerts` itself. A parameter of type `^SList` would only receive a copy of its address; reassigning that parameter would not update the caller's pointer.

The caller must initialize `alerts` to `nil`. The function returns `-1` without changing it if the output parameter is `nil` or the pointer slot is already occupied, avoiding overwriting an existing list. On success, the caller owns the new list and its nodes, even when no readings match and the list is empty. Call `Clear` before deleting the list, because deleting an `SList` alone does not free its nodes.

2.15. Objects and Properties

An **object** keeps data and the functions that work on it together. Its data is stored in **fields** (variables that belong to the object), and its functions are called **methods**. Use objects for things that have their own state and behavior, such as a counter, a document, a sensor, or a network connection.

Object type names conventionally start with a capital `O`, as in `OCounter`.

```
use print

object OCounter:
    value : int = 0

    func Increment():
        value += 1
    endfunc
endobject

func *Main():
    var counter <- OCounter()
    counter.Increment()
    counter.Increment()
    PrintLn("{}", [counter.value])
endfunc
```

This prints `2`. The arrow `<-` creates an **embedded object** named `counter` (an object allocated directly in the local function rather than separately on the heap), and `OCounter()` creates and initializes it. When `Main` finishes, the object is cleaned up automatically.

Objects differ from structures in several important ways:

Feature	Structure	Object
Assignment	Copies the fields	Copies the reference; both variables point to the same object
Allowed fields	Plain data only	Any type, including `str`, `[*]T`, and other objects
Setup and cleanup	None	Constructor (`*Create`), destructor (`*Destroy`), automatic field cleanup
Methods	Plain methods	Supports virtual methods and properties
Type testing	No	Can test the actual type at runtime with `is`

2.15.1 Fields and Methods

An object declaration defines two things between `object Name:` and `endobject`: - **Fields:** variables that belong to the object and hold its data. - **Methods:** functions that belong to the object and can read or update its fields.

A field can have an initial value (such as `lives : int = 3`), which is given to every newly created object. Fields without an initializer start with their zero or empty value: numeric fields start at zero, strings and dynamic arrays start empty, and object references start as `nil` (referring to no object).

```
struct SPoint:
    x : int
    y : int
endstruct

object OPlayer:
    name     : str        // starts empty
    scores   : [*]int     // dynamic array, starts empty
    position : SPoint     // all fields zero
    lives    : int = 3
    rival    : OPlayer    // object reference, starts nil

    func AddScore(points : int):
        scores.Append(points)
    endfunc

    func Total() -> int:
        result = 0
        for points : int in scores:
            result += points
        endfor
```

```
    endfunc
endobject
```

Use a dot (`.`) to access a field or call a method:

```
var player <- OPlayer()
player.name = "Lin"
player.AddScore(10)
player.AddScore(25)
var total : int = player.Total() // 35
```

Inside a method, the object's own fields and methods can be used directly by name—no special prefix is needed.

If an object needs to refer to itself—for example, to pass itself as an argument to another function—it uses the special name `self`, as in `engine.Create(self)`.

DQ checks code from top to bottom. This means a method must be declared before other methods in the same object can call it. If method `A` calls method `B`, write `B` above `A`, or declare the method first and define its body later.

To call a function outside the object from inside a method, use a module prefix: `@.PrintLn(...)` reaches `PrintLn` imported into the current module, while `@print.PrintLn(...)` names the `print` module explicitly. Compiler built-ins such as `SizeOf()` need no prefix. The full lookup rules and method-local `use` are covered under Organizing Object Code.

2.15.2 Object References

An object variable holds a **reference** (a handle pointing to the object in memory), rather than containing a separate copy of the fields. Assigning an object variable to another variable makes both refer to the same object:

```
var counter <- OCounter()
var alias : OCounter = counter
alias.Increment()
// counter.value is now 1: counter and alias refer to the exact same object
```

This is different from structures: copying a structure copies all of its data, so changing one copy leaves the other unchanged.

The same applies when passing objects to functions. An ordinary object parameter receives a reference to the caller's object, so the function can read and modify the object directly without needing a `ref` parameter:

```
func GiveBonus(player : OPlayer):
    player.AddScore(5) // modifies the caller's player object
endfunc
```

An ordinary reference such as `alias` or `player` only **borrows** the object: it can use the object, but it does not own it and is not responsible for destroying it.

2.15.2.1 The `nil` Reference

An object reference can be `nil`, meaning it currently points to no object at all. Like any other local variable in DQ, an object variable must be initialized before it is read (for example, initialized to `nil`).

Always check whether a reference might be `nil` before using it:

```
var selected : OPlayer = nil
// ...
if selected <> nil:
    selected.AddScore(1)
endif
```

Accessing a field or calling a method through a `nil` reference is invalid and stops the program with an error (a segmentation fault).

2.15.3 Creating and Destroying Objects

Every object needs memory to hold its fields. DQ provides three ways to store and manage an object, depending on how long it needs to exist:

Form	How it works	When to use it
`counter <- OCounter()`	**Embedded object:** stored directly in a function's local storage, the global data segment, or a containing object.	Use for local objects, global objects, or fields without a separate heap allocation. Local objects and fields are destroyed automatically; globals are not.
`var counter = new autofree OCounter()`	**Auto-free object:** lives in dynamic memory (the heap). The variable owns it and releases it automatically.	The object needs to outlive a single function, but you still want automatic cleanup without calling `delete`.
`var counter = new OCounter()`	**Manually managed heap object:** lives on the heap. You must destroy it yourself with `delete`.	You want full manual control over when the object is destroyed.

2.15.3.1 Embedded Objects (`<-`)

An **embedded object** declared with `<-` has its storage directly in the context where it is declared:

- Inside a function, it lives in local storage (the stack) and is destroyed automatically when its scope ends, including on an early `return` or when an exception leaves the scope.
- At module scope, it lives in the global data segment. Global embedded objects are not destroyed automatically when the program finishes.
- As a field of another object, it lives inside the containing object and is destroyed with it.

An embedded object cannot be redirected to point to another object, and it cannot be passed to `delete`.

For an embedded field, the constructor call can be delayed: write `field <- OType ..`, without `()`, and call `field.Create(...)` explicitly from the containing object's constructor. See Embedding Objects in Fields for an example.

2.15.3.2 Constructors and Destructors

A **constructor** is a special method named `*Create`. It runs automatically when a new object is being created, allowing you to initialize fields and accept arguments from the caller.

A **destructor** is a special method named `*Destroy`. It runs automatically when the object is being destroyed, allowing you to perform any final cleanup (such as closing a file or releasing an external system resource).

The asterisk `*` marks them as special functions. When calling them explicitly or referring to them, write their names without `*`.

```
use print

object OConnection:
    host : str

    func *Create(ahost : strslice):
        host = ahost
        @.PrintLn("Connected to {}", [host])
    endfunc

    func Send(message : strslice):
        @.PrintLn("{} <- {}", [host, message])
    endfunc

    func *Destroy():
        @.PrintLn("Disconnected from {}", [host])
    endfunc
endobject

func *Main():
    var connection <- OConnection("example.com")
    connection.Send("hello")
endfunc
```

Output:

```
Connected to example.com
example.com <- hello
Disconnected from example.com
```

In `OConnection("example.com")`, the argument `"example.com"` is passed to `*Create`. If an object does not declare a constructor, you create it with empty parentheses: `OType()`.

2.15.3.3 Heap Objects and Autofree Variables (`new` and `autofree`)

The **heap** is general memory where values can live independently of any single function call. Using `new` allocates memory for the object on the heap.

Adding `autofree` gives the variable an **autofree type**. An **autofree variable** destroys its heap object automatically when the variable goes out of scope, when it is set to `nil`, or when a new object is assigned to it. The variable is responsible for releasing the object, so it is also called its **owner**:

```
var counter = new autofree OCounter()
counter.Increment()
counter = new OCounter() // destroys the first counter before taking the new one
counter = nil            // destroys the second counter
```

Without `autofree`, you must manually pair `new` with `delete`:

```
var counter = new OCounter()
counter.Increment()
delete counter = nil // destroys the object, releases its storage, and clears the variable
```

Forgetting to call `delete` leaves the object stranded in memory (a **memory leak**). On the other hand, calling `delete` while other variables still point to the object leaves those variables pointing to invalid memory, which is dangerous (see dangling pointers).

Global objects are not destroyed automatically when the program finishes. See Memory Management and Autofree for details on transferring ownership, autofree array elements, and `ref autofree` parameters.

2.15.3.4 Automatic Field Initialization and Cleanup

Fields are initialized before the constructor body runs, so the constructor can safely use them immediately.

When an object is destroyed, the destructor body runs first. After that, DQ automatically cleans up the object's fields in reverse declaration order: - Managed strings (`str`) and dynamic arrays release their allocated storage. - Embedded objects are destroyed. - Autofree fields destroy the heap objects they hold.

Because of this automatic cleanup, you only need to write a `*Destroy` method when you have custom cleanup work to do, such as closing external handles or logging a message. Ordinary object references in fields are just borrows, so the objects they point to are not destroyed.

If an error or exception occurs inside a constructor, the object creation fails. Any fields and base parts that were already set up are cleaned up in reverse order, so half-constructed objects are never left behind. See Exceptions and Error Handling.

2.15.3.5 Overloaded Constructors

You can provide more than one constructor by adding the `[[overload]]` attribute to each. This gives callers different ways to create the object. One constructor can also call another constructor to reuse setup code:

```
object OColor:
    red   : byte
    green : byte
    blue  : byte

    func *Create(ared : byte, agreen : byte, ablue : byte) [[overload]]:
        red = ared
        green = agreen
        blue = ablue
    endfunc

    func *Create(gray : byte) [[overload]]:
```

```
        Create(gray, gray, gray) // calls the three-argument constructor
    endfunc
endobject

var orange <- OColor(255, 165, 0)
var gray <- OColor(128) // red, green, and blue are all 128
```

2.15.3.6 Embedding Objects in Fields

Just as a structure can contain another structure, an object can contain another object as one of its fields. Declare the embedded object with `<-` to construct it as part of the containing object and destroy it with that object:

```
object OEngine:
    power : int

    func *Create(apower : int):
        power = apower
    endfunc
endobject

object OCar:
    engine <- OEngine(120) // created with every OCar, destroyed with it
endobject
```

Sometimes the embedded object needs arguments that are only available when the outer object's constructor runs. In that case, omit the parentheses and write `..` after the type name (`engine <- OEngine ..`). This reserves the storage without calling the constructor. Call `Create` explicitly inside the outer constructor:

```
object OCar:
    engine <- OEngine .. // reserves storage; constructed in OCar.Create

    func *Create(apower : int):
        engine.Create(apower)
    endfunc
endobject
```

An embedded object must be constructed before code attempts to use it. Its storage belongs entirely to the containing object, so never pass an embedded object to `delete`. The composed objects example shows how to use this pattern to give an embedded object a reference back to its container.

2.15.4 Member Visibility

By default, all members of an object (its fields, methods, and properties) are **public**, meaning any code that uses the object can access them.

You can restrict access using visibility labels:

Label	Who can access the members that follow
`public`	Any code. This is the default.
`protected`	Only methods of this object and of objects derived from it.
`private`	Only methods of this specific object.

Visibility labels act like section headers inside the object: all members declared below a label share that visibility until another label appears.

Use visibility labels to protect an object's internal data from unauthorized or accidental changes. In the following example, callers can inspect the balance, but can only change it through `Deposit`, which ensures invalid amounts are rejected:

```
object OAccount:
private
    balance : int = 0
public
    func GetBalance() -> int:
        return balance
    endfunc

    func Deposit(amount : int) -> bool:
        if amount <= 0:
            return false
```

```
        endif
        balance += amount
        return true
    endfunc
endobject
```

Code using an `OAccount` can call `account.GetBalance()`, but cannot touch `balance` directly. The next section introduces properties, which let callers read a value with field syntax while keeping the underlying field private.

2.15.5 Properties

A **property** looks and acts like a regular field to anyone using the object, but behind the scenes it can use methods to read or write the value.

Properties give you the best of both worlds: clean, natural syntax (such as `volume.value = 80`) combined with the ability to validate input, compute values on demand, or expose read-only data while keeping internal fields private.

2.15.5.1 Field-Backed Properties

The simplest property reads from or writes to a private field directly. Omitting `write` makes it a **read-only property**:

```
object OTimer:
private
    elapsed : int = 0
public
    property seconds : int read elapsed // read-only for outside callers

    func Tick():
        elapsed += 1
    endfunc
endobject
```

Outside callers can read `timer.seconds`, but attempting to assign to it (`timer.seconds = 0`) is caught at compile time with an error.

2.15.5.2 Computed Properties

A **getter** method computes the value each time the property is read, without needing to store it in a separate field:

```
object OTemperature:
public
    celsius : float = 20
private
    func GetFahrenheit() -> float:
        return celsius * 9 / 5 + 32
    endfunc
public
    property fahrenheit : float read GetFahrenheit
endobject

var temp <- OTemperature()
temp.celsius = 25
var f : float = temp.fahrenheit // computes 77 dynamically
```

No `fahrenheit` field is stored in memory; it always calculates the temperature from the current `celsius` value.

2.15.5.3 Checking New Values with Setters

A **setter** method receives the new value and checks or adjusts it before storing it:

```
use print

object OVolume:
private
    level : int = 50

    func SetLevel(value : int):
        if value < 0:
            level = 0
        elif value > 100:
            level = 100
        else:
            level = value
        endif
    endfunc
public
```

```
    property value : int read level write SetLevel
endobject

func *Main():
    var volume <- OVolume()
    volume.value = 150 // SetLevel limits it to 100
    volume.value -= 10 // reads 100, then calls SetLevel(90)
    PrintLn("{}", [volume.value])
endfunc
```

This prints `90`.

2.15.5.4 Property Accessor Rules

The methods used to read and write a property are called **accessors**: - `read` specifies the field or getter method that supplies the value. A getter takes no parameters and returns the property type. - `write` specifies the field or setter method that updates the value. A setter takes the new value as its only parameter and returns nothing.

Accessor methods must belong to the same object and must be declared before the property that uses them.

A property needs at least one accessor: - Omit `write` for a read-only property. - Omit `read` for a write-only property. - Compound assignment operators such as `+=` or `-=` read the current value, modify it, and write it back, so they require both `read` and `write`.

2.15.5.5 Indexed and Default Properties

An **indexed property** lets callers use square bracket syntax (`[]`), similar to an array. Its getter receives the index; its setter receives the index followed by the new value:

```
object OScores:
private
    values : [3]int

    func GetScore(index : int) -> int:
        return values[index]
    endfunc

    func SetScore(index : int, value : int):
        values[index] = value
    endfunc
public
    property score : [index : int]int default read GetScore write SetScore
endobject

var scores <- OScores()
scores.score[0] = 10
scores[1] = 20 // the default keyword lets you omit .score
```

Indices are accessor parameters and are not restricted to integers. For example, `OStrStrMap` in `collections/strstrmap` declares its `item` property with a string key:

```
property item : [akey : str]str default read GetItem write SetItem
```

Callers can use a string with either the property name or the default syntax:

```
use collections/strstrmap, print

func *Main():
    var settings <- OStrStrMap()
    settings.item["theme"] = "dark"
    PrintLn("{}", [settings["theme"]]) // prints dark
endfunc
```

An indexed property can accept more than one index, which is useful for grids and tables:

```
property cell : [row : int, col : int]int read GetCell write SetCell
```

Callers can use this with multiple indices: `grid.cell[1, 2] = 5`.

Putting `default` after the indexed property type allows callers to use brackets directly on the object itself (such as `scores[1]` instead of `scores.score[1]`). When an object and its base types declare several default properties, the one declared latest is selected.

2.15.5.6 Property Restrictions

A property is not an actual variable stored at a single memory address: reading it calls a getter or reads a field, and writing it calls a setter. Because of this:

- A property cannot be passed to a `ref`, `refin`, or `refout` parameter. To pass a property's value to a function that modifies it through `ref`, copy the value into a temporary local variable, pass that variable, and assign the updated value back to the property:

```
var level : int = volume.value
Adjust(level) // func Adjust(value : ref int)
volume.value = level
```

- You cannot take the memory address of a property using `%`.

Values returned by a property behave normally; for example, a property that returns an object reference can be followed immediately by member access, as in `window.current_button.Click()`.

2.15.6 Inheritance

Inheritance allows a new object type to build on an existing object type: - The existing type is called the **base object** (or parent). - The new type is called the **derived object** (or child).

The derived object automatically inherits all fields and methods from the base object and can add its own new fields and methods. In DQ, an object can inherit from one base object, written in parentheses after its name:

```
object OAnimal:
    name : str

    func *Create(aname : strslice):
        name = aname
    endfunc
endobject

object ODog(OAnimal):
    tricks : int = 0

    func Learn():
        tricks += 1
    endfunc
endobject

var dog <- ODog("Rex") // uses the inherited OAnimal constructor
dog.Learn()
var animal : OAnimal = dog // a derived object can be assigned to a base reference
```

A derived object can be used wherever its base type is expected, as shown in the assignment `var animal : OAnimal = dog`. When viewed through an `animal` reference, only the members declared in `OAnimal` are accessible.

2.15.6.1 Constructors and Inheritance

When an object inherits from a base type, the base part must be initialized:

- If the derived object does not declare a constructor of its own, it automatically inherits the base constructor (as `ODog("Rex")` does above).
- If the derived object declares its own constructor, it must call the base constructor using `inherited Create(...)`:

```
object OGuideDog(ODog):
    owner : str

    func *Create(aname : strslice, aowner : strslice):
        inherited Create(aname) // initialize the base OAnimal/ODog part first
        owner = aowner
    endfunc
endobject
```

If the base constructor takes no parameters, calling `inherited Create()` is optional; DQ calls it automatically if you omit it. If the base constructor requires parameters, the derived constructor must call it explicitly. The compiler prevents calling the base constructor more than once.

Object setup runs from base to derived, and cleanup runs in reverse order: when an object is destroyed, the derived destructor runs first, and then the base destructor runs automatically. A destructor may call `inherited Destroy()` explicitly if the base cleanup must happen at a specific point during destruction.

2.15.6.2 Virtual Methods

Normally, when you call a method through an object reference, the compiler decides which method to run based on the **declared type** of the variable. For example, if a variable is declared as `var animal : OAnimal`, calling `animal.Sound()` would normally call `OAnimal.Sound()`, even if the variable currently holds an `ODog`.

A **virtual method** changes this: it tells DQ to inspect the **actual object** in memory at runtime and run that specific object's version of the method instead.

- In the base object, mark the method with `[[virtual]]` to allow derived objects to customize it.
- In the derived object, mark the replacement method with `[[override]]`.

```
use print

object OAnimal:
    name : str

    func *Create(aname : strslice):
        name = aname
    endfunc

    func Sound() -> str [[virtual]]:
        return "..."
    endfunc

    func Speak():
        @.PrintLn("{} says {}", [name, Sound()])
    endfunc
endobject

object ODog(OAnimal):
    func Sound() -> str [[override]]:
        return "Woof"
    endfunc
endobject

object OCat(OAnimal):
    func Sound() -> str [[override]]:
        return "Meow"
    endfunc
endobject

func *Main():
    var dog <- ODog("Rex")
    var cat <- OCat("Tom")
    var animals : [2]OAnimal = [dog, cat]
    for animal :? in animals:
        animal.Speak()
    endfor
endfunc
```

Output:

```
Rex says Woof
Tom says Meow
```

`Speak` is declared only in `OAnimal`, yet its call to `Sound()` runs the derived implementation for each animal. Without `[[virtual]]`, both lines would print `...`.

The override must have the exact same parameters and return type as the virtual base method. The compiler reports an error if `[[override]]` is used on a method that has no matching virtual method in the base type, or if a derived method has the same name as a virtual method but forgets `[[override]]`.

Accessor methods can be virtual. A derived object can override an inherited accessor method to customize how the property behaves.

A destructor cannot call virtual methods, because derived parts of the object may already have been torn down.

2.15.6.3 Calling the Base Implementation

Inside an overriding method, `inherited` calls the base type's version of that method. This lets you extend the base behavior rather than replacing it entirely:

```
object OLoudDog(ODog):
    func Sound() -> str [[override]]:
        var base : str = inherited Sound()
        return base + "!!!" // "Woof!!!"
    endfunc
endobject
```

Use the explicit call syntax, `inherited Sound()`, when you need the return value or want to pass different arguments. A bare `inherited` statement calls the base method with the current arguments and ignores any return value.

2.15.6.4 Abstract and Final Methods

An **abstract method** has a header but no body. It acts as a requirement: the base object declares that every derived object *must* provide its own version of this method. Declare it with `[[virtual, abstract]]` and two dots `..` instead of a body:

```
object OShape:
    func Area() -> float [[virtual, abstract]] ..
endobject
```

Because an abstract object type has incomplete methods, **you cannot create an object directly from an abstract type** (writing `new OShape()` or `<- OShape()` is an error). You can only create objects from derived types that have implemented all abstract methods.

A **final method**, marked with `[[virtual, final]]`, prevents any further derived types from overriding it.

2.15.7 Runtime Types

2.15.7.1 Runtime Type Tests and Casts

When you hold a reference of a base type (such as `var animal : OAnimal`), it might point to an `ODog`, an `OCat`, or another derived object. DQ provides two tools to check and convert references safely at runtime:

2.15.7.1.1 TESTING TYPES WITH `is`

The `is` operator tests whether an object reference points to an object of a particular type or one of its derived types. It returns `true` or `false`, and safely returns `false` if the reference is `nil`:

```
var dog <- ODog("Rex")
var animal : OAnimal = dog
if animal is ODog:
    // true: animal refers to an ODog
endif
```

2.15.7.1.2 CONVERTING REFERENCES WITH `TryCast()`

To access fields or methods that exist only on the derived type, you must convert the base reference to the derived type. This conversion is called a **type cast**.

Because the object might not actually be of that type, use `TryCast(TargetType, source)` for a safe, checked conversion. Local variables can be declared in any of these three forms:

- `var specific : ODog = TryCast(ODog, animal)`
- `var specific :? = TryCast(ODog, animal)`
- `var specific = TryCast(ODog, animal)`

The last two forms infer the variable's type from the target type passed to `TryCast`. All three forms check the actual object at runtime:

- If the object is an `ODog` (or a type derived from `ODog`), the new variable receives a reference to that object.

- If the object is something else (such as an `OCat`), or if `animal` is `nil`, the new variable receives `nil` instead of crashing.

```
var asdog = TryCast(ODog, animal) // the ODog, or nil
var ascat = TryCast(OCat, animal) // nil: animal is not an OCat
if asdog <> nil:
    asdog.Learn()
endif
```

Converting a reference only changes how the compiler views the object. It does not copy the object, and it does not transfer ownership.

2.15.7.2 Object Type Variables and Type Names

In DQ, variables can hold data values, but they can also hold *types*. A `type of OBase` variable can store the type `OBase` or any type derived from it. This allows a program to choose at runtime which kind of object to construct:

```
func MakeAnimal(cat : bool, name : strslice) -> OAnimal:
    var animaltype : type of OAnimal = ODog
    if cat:
        animaltype = OCat
    endif
    return new animaltype(name) // creates an ODog or OCat using the OAnimal constructor
endfunc
```

Calling `new` on the type variable constructs an object of whichever type was stored in it. The selected type must accept the constructor arguments of the base type. Unrelated types, abstract types, and types with incompatible constructors are rejected by the compiler. The returned heap object still needs an autofree variable or an explicit `delete`.

2.15.7.2.1 QUERYING TYPE NAMES

`TypeName(Type)` returns the name of a type as text. When passed an object reference, `TypeName(reference)` returns the actual runtime type of the object if it is non-`nil`, or the variable's declared type if it is `nil`:

```
var animal : OAnimal = MakeAnimal(true, "Tom")
PrintLn("{}", [TypeName(animal)]) // prints "OCat"
delete animal
```

Use `TypeName` for debugging and logging rather than as a permanent identifier in file formats or network protocols.

2.15.8 Object Method References

DQ allows you to store a function in a variable so it can be called later (a function reference). Because a method belongs to a specific object instance, storing a method requires an **object method reference**, written as `func(...) of object`.

An object method reference bundles two things together:

1. The method to call.
2. The specific object instance to call it on.

When you assign a virtual method to an object method reference, DQ resolves the method implementation for that specific object at the time of assignment. The reference only borrows the object—it does not keep the object alive. See Object Method References for a complete example.

2.15.9 Organizing Object Code

As objects grow, method bodies can be defined separately from their declarations. Forward declarations let object types refer to each other, and explicit module prefixes or method-local `use` control access to outside names.

2.15.9.1 Declaring Methods Separately

Every method has two parts:

1. **The declaration (the header):** tells the compiler the method's name, its parameters, and what type it returns.
2. **The definition (the body):** the actual lines of code that run when the method is called.

Both parts can be written together directly inside the `object ... endobject` block. However, when an object has many methods or long code, keeping everything inside the object can make it difficult to read. DQ lets you separate them:

- Put just the **declaration** inside the object, followed by `..` (two dots) to indicate that the body will be provided later.
- Write the full **definition** (the body) outside the object, prefixing the method name with the object type name, such as `func OButton.Click():`.

```
object OButton:
    caption : str

    func Click() ..
    func *Create(acaption : strslice) ..
endobject

func OButton.*Create(acaption : strslice):
    caption = acaption
endfunc

func OButton.Click():
    @.PrintLn("{} clicked", [caption])
endfunc
```

The definition outside the object must match the declaration's parameters, return type, and attributes. Inside the separated method body, all fields and methods of the object are available directly by name, exactly as if the method were written directly inside the object block.

2.15.9.2 Forward Declarations

The DQ compiler reads source files from top to bottom. If two object types reference each other (for example, `OParent` has a field pointing to `OChild`, and `OChild` has a field pointing to `OParent`), one of them must be introduced before its full description is written.

A **forward declaration** introduces the name of an object type using `..` so other declarations can refer to it:

```
object OParent .. // forward declaration: introduces OParent

object OChild:
    parent : OParent // allowed because OParent was introduced above
endobject

object OParent:
    child : OChild
endobject
```

The full definition of the object must be supplied before any code creates the object, accesses its fields, or calls its methods.

2.15.9.3 Accessing Variables and Functions Outside of the Object

Inside an object method, bare names always refer to the object's own members (its fields, methods, and properties). Variables, constants, and functions outside of the object scope are **not** directly available by their bare names. This restriction applies to:

- Global variables and constants declared in the same module.
- Functions declared in the same module.
- Symbols from imported modules, including the runtime's built-in `sys` module (such as `MemSet`, `MemCopy`, or `MemAlloc`).

This scope separation prevents accidental naming conflicts—an object's field (such as `count`, `status`, or `length`) can never be confused with an outside variable or library function with the same name.

The exception to this rule is **compiler internal built-ins**, such as `SizeOf()`, `OffsetOf()`, and `TypeName()`, as well as language types (such as `int`, `float`, `str`, `rostr`, or `SPoint`). These are built into the language itself and are always available everywhere without any prefix.

To access outside variables, constants, or functions from inside a method, you normally prefix them with `@`:

- **`@module.Name` (fully qualified):** names the imported module and the symbol, such as `@print.PrintLn`, `@sys.MemSet`, or `@socket.AF_INET`.
- **`@..Name` (current module only):** refers to a symbol that is defined in the current module (until this source code position)

- `@.Name` **(full outside scope):** the short form. It reaches everything available in the current module at that source code position, including its own global variables and constants, as well as imported functions.

```
use print

var greeting_count : int = 0

object OGreeter:
    name : str = "Lin"

    func Greet():
        @print.PrintLn("Hello, {}!", [name]) // fully qualified module reference
        @.greeting_count += 1                // short form for a module variable
        @.PrintLn("Greetings so far: {}", [@.greeting_count])
    endfunc
endobject
```

Writing `PrintLn(...)` or `greeting_count` without a prefix inside `Greet` is a compile error (`ObjVsInModuleNs`), and the compiler error message suggests the prefixed form.

2.15.9.4 Method-Local Use

While prefixing outside symbols with `@.` or `@module.` works well for occasional calls, practical code often uses many outside definitions at once. For example, low-level networking, file I/O, or system routines frequently rely on dozens of constants, types, and functions from modules such as `socket` or `sys` . Writing `@socket.AF_INET` , `@socket.libc_htons(...)` , and `@sys.MemSet(...)` on every single line quickly becomes verbose and clutters the code.

To solve this, DQ provides the **method-local** `use` statement. Placing a `use` statement at the very beginning of a method body opens outside namespaces for direct, unqualified access inside that specific method:

```
func ONanoDatagram.SetRemoteAddr(aaddr : rostr, aport : uint16):
    use socket, sys
    MemSet(%remote_addr, 0, SizeOf(remote_addr))
    remote_addr_len = SizeOf(remote_addr)
    remote_addr.sin_family = AF_INET
    remote_addr.sin_port = libc_htons(aport)

    if libc_inet_pton(AF_INET, aaddr, %remote_addr.sin_addr) <> 1:
        raise ENanoNet('Invalid Host IP address: {}', [aaddr])
    endif
endfunc
```

In this method:

- `SizeOf(remote_addr)` needs no prefix because `SizeOf()` is a compiler internal function available everywhere.
- `MemSet` comes from the `sys` module, while `AF_INET` , `libc_htons` , and `libc_inet_pton` come from `socket` .
- The `use socket, sys` statement makes all of them accessible directly by their bare names throughout the method body.

2.15.9.4.1 FORMS OF METHOD-LOCAL `use`

Method-local `use` supports the following forms:

- `use module1, module2` **:** Opens one or more specific imported module namespaces for direct access (as shown with `use socket, sys` above).
- **Using an imported alias:** If a module was imported at the file level with an alias (such as `use libc/stdio as cstdio`), you must use that **alias name** in the method-local statement: `dq`

  ```
  func OLogger.Log(msg : strslice):
      use cstdio // refers to libc/stdio via its module alias
      printf("%s\n", msg.pchar)
  endfunc
  ```

- `use .` **(local module definitions):** Makes all declarations belonging to the current module itself (its global variables, constants, and functions) directly available without `@.` prefixes.
- `use *` **(make everything available):** Makes *all* symbols visible at the current module level directly available inside the method, including the current module's own declarations and all imported modules.

2.15.9.4.2 SCOPE AND RULES

- Method-local `use` only affects name lookup **inside that specific method body**. It does not affect any other methods in the same object or file.
- It does not load new modules from disk; it only opts already imported module namespaces into unqualified lookup within the method.
- Object members always take precedence: if an object field has the same name as an outside symbol brought in by `use`, using that name inside the method accesses the object's field. To reach the outside symbol in that case, use an explicit `@module.Name` or `@.Name` prefix.

2.15.10 Object Examples

2.15.10.1 A Short Object Demo

This example combines inheritance, a read-only property, an embedded helper, and autofree variables. `OChild` changes a protected field and extends a virtual method, while `OBase` manages the helper's construction and cleanup.

```
use print

object OBase ..

object OHelper:
    base : OBase

    func *Create(abase : OBase) ..
    func *Destroy() ..
endobject

object OBase:
protected
    intval : int = 1
public
    name   : embstr(64)
    helper <- OHelper ..

    property value : int  read intval

    func PrintExtra()  [[virtual]]:
        @.Print(' value={}', [value])
    endfunc

    func *Create(aname : rostr):
        name = aname
        helper.Create(self)
    endfunc

    func *Destroy():
        @.PrintLn('"{}" destroyed', [name])
    endfunc

    func Print()  [[virtual]]:
        @print.Print('{}:', [name])
        PrintExtra()
        @.PrintLn()
    endfunc
endobject

func OHelper.*Create(abase : OBase):
    base = abase
    @.PrintLn('Helper for "{}" created', [base.name])
endfunc

func OHelper.*Destroy():
    @.PrintLn('Helper for "{}" destroyed', [base.name])
endfunc


object OChild(OBase):
    flval : float

    func *Create(aname : rostr, aflval : float):
        inherited Create(aname)
        intval = 2
        flval = aflval
    endfunc

    func PrintExtra()  [[override]]:
        inherited
        @.Print(' flval={}', [flval])
    endfunc
endobject

func Manipulate(aobj : ref autofree OBase):
```

```
    aobj = new OChild('Child3-heap', 7 / 3)
    aobj.Print()
endfunc

func ErrorTest():
    PrintLn('Error test')
    try:
        var obj = new autofree OChild('ErrChild-heap', 8.3)
        raise Exception('Some error happened!')
    except Exception as e:
        PrintLn('ERROR: {}', [e.message])
        e.PrintBacktrace()
    endtry
endfunc

func *Main():
    PrintLn('DQ Objects Demo')

    var ob = new autofree OBase('Base-heap')
    ob.Print()

    if true:
        var oc1 <- OChild('Child1-stack', 3 / 2)
        oc1.Print()
    endif

    ErrorTest()

    PrintLn('reassigning autofree type')
    ob = new OChild('Child2-heap', 3.14)
    ob.Print()

    PrintLn('Autofree type as argument')
    Manipulate(ob)

    PrintLn('nil-ing autofree type')
    ob = nil

    PrintLn('Test finished.')
endfunc
```

Source: dq_intro_3.dq.

This test outputs:

```
DQ Objects Demo
Helper for "Base-heap" created
Base-heap: value=1
Helper for "Child1-stack" created
Child1-stack: value=2 flval=1.5
"Child1-stack" destroyed
Helper for "Child1-stack" destroyed
Error test
Helper for "ErrChild-heap" created
"ErrChild-heap" destroyed
Helper for "ErrChild-heap" destroyed
ERROR: Some error happened!
Backtrace =
  #0 ErrorTest @ /workvc/dq-lang/examples/dq_intro_3.dq:75
  #1 Main @ /workvc/dq-lang/examples/dq_intro_3.dq:93
reassigning autofree type
"Base-heap" destroyed
Helper for "Base-heap" destroyed
Helper for "Child2-heap" created
Child2-heap: value=2 flval=3.14
Autofree type as argument
"Child2-heap" destroyed
Helper for "Child2-heap" destroyed
Helper for "Child3-heap" created
Child3-heap: value=2 flval=2.333333333333334
nil-ing autofree type
"Child3-heap" destroyed
Helper for "Child3-heap" destroyed
Test finished.
```

`OChild` can change the protected `intval` field, while outside callers can only read it through the inherited `value` property. Calling `Print()` runs `OBase.Print`, whose virtual call to `PrintExtra()` selects the child's override. The bare `inherited` in that override prints the base value before the child adds `flval`.

`helper.Create(self)` gives the embedded helper a borrowed reference to its container. The helper's method bodies appear after the full `OBase` definition so they can access `base.name`. The name uses `embstr(64)`, keeping its text storage inside the object. Because `helper` is declared after `name`, it is destroyed first and can still read the name during cleanup.

`ErrorTest` shows that the local autofree variable destroys its object before the exception handler runs. The handler then calls `e.PrintBacktrace()`; its frames depend on the build and source location. The rest of `Main` demonstrates autofree variable replacement, replacement through a `ref autofree OBase` parameter, and early destruction by assigning `nil`.

2.15.10.2 A Temperature Log with Properties

`OTemperatureLog` keeps a limited number of recent temperature readings. Its internal storage is private; callers interact with it exclusively through properties and methods:

Property	Kind	Purpose
`count`	read-only, computed	Current number of stored readings
`average`	read-only, computed	Average of all stored readings
`capacity`	field-backed read, validated setter	Maximum number of readings; lowering it discards the oldest extra readings
`reading`	indexed, default	Reads or updates a stored reading by index

```
use print

object OTemperatureLog:
private
    readings : [*]float
    limit    : int = 5

    func GetCount() -> int:
        return readings.length
    endfunc

    func GetAverage() -> float:
        result = 0
        for value : float in readings:
            result += value
        endfor
        if readings.length > 0:
            result /= readings.length
        endif
    endfunc

    func SetCapacity(value : int):
        limit = iif(value < 1, 1, value)
        while readings.length > limit:
            readings.Delete(0) // drop the oldest readings
        endwhile
    endfunc

    func GetReading(index : int) -> float:
        return readings[index]
    endfunc

    func SetReading(index : int, value : float):
        readings[index] = value
    endfunc

public
    property count    : int   read GetCount
    property average  : float read GetAverage
    property capacity : int   read limit write SetCapacity
    property reading  : [index : int]float default read GetReading write SetReading

    func Add(value : float):
        if readings.length == limit:
            readings.Delete(0)
        endif
        readings.Append(value)
    endfunc

    func Print():
        @.Print("count {}:", [count])
        for i : int = 0 count count:
            @.Print(" {}", [reading[i]])
        endfor
        @.PrintLn(", average {:.2f}", [average])
    endfunc
endobject

func *Main():
    var log <- OTemperatureLog()
    log.Add(21.5)
    log.Add(22)
    log.Add(35)
```

```
    log.Add(22.5)
    log.Print()

    log[2] = 23 // corrects a wrong reading through the default property
    log.reading[0] += 0.5
    log.Print()

    log.capacity = 2 // keeps only the two newest readings
    log.Print()

    log.capacity = -5 // SetCapacity limits it to a minimum of 1
    log.Add(24)
    log.Print()
    PrintLn("Capacity: {}", [log.capacity])
endfunc
```

Output:

```
count 4: 21.5 22 35 22.5, average 25.25
count 4: 22 22 23 22.5, average 22.38
count 2: 23 22.5, average 22.75
count 1: 24, average 24.00
Capacity: 1
```

`count` and `average` do not have dedicated storage fields; their getters compute fresh values directly from the `readings` array each time they are read. `capacity` reads the private `limit` field directly, but writes go through `SetCapacity`, which enforces a minimum value of 1 and trims excess elements.

`log[2] = 23` invokes `SetReading(2, 23)` through the default property. `log.reading[0] += 0.5` exercises both accessors: it calls `GetReading(0)`, adds `0.5`, and then passes the result to `SetReading(0, 22)`. Inside `Print`, properties are accessed naturally by name, such as `count` and `reading[i]`.

Because `readings` is private, external code cannot modify the array directly or bypass the validation logic in `Add` and `SetCapacity`.

2.15.10.3 Shapes in an Array with Autofree Elements

This example builds a shape hierarchy. `OShape` declares an abstract `Area` method, and each derived shape provides its own calculation. A dynamic array of `autofree` references owns the shapes, so they are destroyed automatically when they are removed from the array or when the array leaves scope.

```
use print

object OShape:
    name : str

    func *Create(aname : strslice):
        name = aname
    endfunc

    func Area() -> float [[virtual, abstract]] ..

    func Describe() [[virtual]]:
        @.PrintLn("{}: area {:.2f}", [name, Area()])
    endfunc

    func *Destroy():
        @.PrintLn("Released {}", [name])
    endfunc
endobject

object ORectangle(OShape):
    width  : float
    height : float

    func *Create(awidth : float, aheight : float):
        inherited Create("rectangle")
        width = awidth
        height = aheight
    endfunc

    func Area() -> float [[override]]:
        return width * height
    endfunc
endobject

object OSquare(ORectangle):
    func *Create(aside : float):
        inherited Create(aside, aside)
        name = "square"
    endfunc
endobject
```

```
object OCircle(OShape):
    radius : float

    func *Create(aradius : float):
        inherited Create("circle")
        radius = aradius
    endfunc

    func Area() -> float [[override]]:
        return 3.14159 * radius * radius
    endfunc

    func Describe() [[override]]:
        inherited // prints the name and the area
        @.PrintLn("  radius: {}", [radius])
    endfunc
endobject

func TotalArea(shapes : []autofree OShape) -> float:
    result = 0
    for shape : OShape in shapes:
        result += shape.Area() // calls the Area method of the actual shape type
    endfor
endfunc

func *Main():
    var shapes : [*]autofree OShape // the array owns the shapes
    shapes.Append(new ORectangle(4, 2.5))
    shapes.Append(new OSquare(3))
    shapes.Append(new OCircle(1.5))

    for shape : OShape in shapes:
        shape.Describe()
    endfor
    PrintLn("Total area: {:.2f}", [TotalArea(shapes)])

    var rectangles : int = 0
    for shape : OShape in shapes:
        if shape is ORectangle: // true for OSquare too
            ++rectangles
        endif

        var circle = TryCast(OCircle, shape) // nil for other shapes
        if circle <> nil:
            circle.radius *= 2
            PrintLn("Enlarged {}: area {:.2f}", [TypeName(circle), circle.Area()])
        endif
    endfor
    PrintLn("Rectangles: {}", [rectangles])

    shapes.Delete(0) // destroys the rectangle
    PrintLn("Remaining: {}", [shapes.length])
endfunc
```

Output:

```
rectangle: area 10.00
square: area 9.00
circle: area 7.07
  radius: 1.5
Total area: 26.07
Enlarged OCircle: area 28.27
Rectangles: 2
Released rectangle
Remaining: 2
Released square
Released circle
```

`OSquare` inherits from `ORectangle` and reuses its constructor and `Area` calculation, simply passing the same value for both width and height. `OCircle` overrides `Describe` and uses a bare `inherited` call to run the base shape description before printing its radius.

Inside the loop, `shape` has the base type `OShape`, yet each `shape.Area()` call selects the correct derived formula at runtime. `shape is ORectangle` is also true for the square, because `OSquare` derives from `ORectangle`. `TryCast` safely converts the reference to `OCircle`, returning `nil` for any other shape.

The loop variables and the `TotalArea` parameter only borrow the shapes. The `shapes` dynamic array owns them: `shapes.Delete(0)` immediately destroys the rectangle, and the remaining shapes are destroyed automatically when `shapes` goes out of scope at the end of `Main`.

2.15.10.4 Lifetime of Composed Objects

This example traces how objects are created and destroyed when objects are composed together. An `OMachine` contains an embedded `OEngine`, and the engine keeps an ordinary reference back to its machine. `OPump` inherits from `OMachine` and extends its `Run` method.

```
use print

object OMachine ..

object OEngine:
    machine : OMachine // back-reference; the engine does not own the machine

    func *Create(amachine : OMachine) ..
    func Start() ..
    func *Destroy() ..
endobject

object OMachine:
    name   : str
    engine <- OEngine .. // constructed later, in OMachine.Create

    func *Create(aname : strslice):
        name = aname
        @.PrintLn("{}: created", [name])
        engine.Create(self)
    endfunc

    func Run() [[virtual]]:
        engine.Start()
    endfunc

    func *Destroy():
        @.PrintLn("{}: destroyed", [name])
    endfunc
endobject

func OEngine.*Create(amachine : OMachine):
    machine = amachine
    @.PrintLn("{}: engine created", [machine.name])
endfunc

func OEngine.Start():
    @.PrintLn("{}: engine running", [machine.name])
endfunc

func OEngine.*Destroy():
    @.PrintLn("{}: engine destroyed", [machine.name])
endfunc

object OPump(OMachine):
    rate : int

    func *Create(aname : strslice, arate : int):
        inherited Create(aname)
        rate = arate
    endfunc

    func Run() [[override]]:
        inherited
        @.PrintLn("{}: pumping {} l/min", [name, rate])
    endfunc
endobject

func ReplaceMachine(machine : ref autofree OMachine):
    machine = new OPump("pump-2", 40) // releases the caller's previous machine
endfunc

func *Main():
    PrintLn("-- embedded object in a block")
    if true:
        var local <- OPump("pump-1", 25)
        local.Run()
    endif

    PrintLn("-- autofree variable replacement")
    var current = new autofree OMachine("mixer")
    current.Run()
    ReplaceMachine(current)
    current.Run()

    PrintLn("-- exception")
    try:
        var temporary = new autofree OMachine("tester")
        raise Exception("power failure")
    except Exception as e:
        PrintLn("Error: {}", [e.message])
    endtry
```

```
    PrintLn("-- end of Main")
endfunc
```

Output:

```
-- embedded object in a block
pump-1: created
pump-1: engine created
pump-1: engine running
pump-1: pumping 25 l/min
pump-1: destroyed
pump-1: engine destroyed
-- autofree variable replacement
mixer: created
mixer: engine created
mixer: engine running
mixer: destroyed
mixer: engine destroyed
pump-2: created
pump-2: engine created
pump-2: engine running
pump-2: pumping 40 l/min
-- exception
tester: created
tester: engine created
tester: destroyed
tester: engine destroyed
Error: power failure
-- end of Main
pump-2: destroyed
pump-2: engine destroyed
```

`OEngine` needs to store a reference to its machine, but `OMachine` is declared further down in the file. The forward declaration `object OMachine ..` introduces the type name so `OEngine` can declare the `machine : OMachine` field. The engine's method bodies are defined after `OMachine` is fully declared because they access `machine.name`.

The engine cannot be constructed in its field declaration line because the machine's `self` reference is only available inside `OMachine.Create`. The `..` in `engine <- OEngine ..` reserves the field's storage, delaying construction until `engine.Create(self)` is called.

Each machine's destructor runs before its fields are cleaned up, which is why `destroyed` appears before `engine destroyed`. Furthermore, fields are cleaned up in reverse declaration order, so `engine` is destroyed before `name`, ensuring `machine.name` remains valid while the engine's destructor runs.

The three sections illustrate the different ways objects are cleaned up: - `local` is destroyed when the `if` block ends. - `ReplaceMachine` replaces the caller's autofree variable via a `ref autofree` parameter, which automatically destroys `"mixer"` before assigning `"pump-2"`. - `temporary` is destroyed when the exception escapes the `try` block, before the `except` handler begins executing. - Finally, `current` releases `"pump-2"` when `Main` returns.

2.16. Memory Management and Autofree

The earlier chapters introduced automatic string and array storage, borrowed pointers, and the three ways to create objects. This chapter brings those rules together and explains how **autofree variables** manage heap allocations as values pass between functions, objects, and collections.

An **autofree type**, such as `autofree OTask` or `autofree ^SPoint`, adds automatic cleanup to an object-reference or pointer type. A variable of that type releases its allocation when it is replaced or leaves scope. It is also called the allocation's **owner**: it is responsible for releasing that allocation. A **borrow** lets other code use the allocation without taking responsibility for its cleanup.

Keeping that responsibility clear prevents both memory leaks (storage that is never released) and dangling references (references to storage that no longer exists).

For object construction and destructors, see Creating and Destroying Objects. For the lifetime of an address, see Allocation and Lifetime.

2.16.1 Choosing Storage and Cleanup

Lifetime means how long a value's storage exists. **Ownership** means who is responsible for releasing it. Choose a form that fits how you use the value:

Form	Lifetime and responsibility
Number, fixed array, structure, fixed-storage embedded string	Lives directly in its containing variable or object.
`str`	Manages its text storage; changing a shared string makes an independent value.
`[*]T`	Manages shared array storage; aliases see changes to the same array.
`var task <- OTask(...)`	Embedded object; lives with its scope or containing object.
`new OTask(...)`	Heap object; needs an explicit `delete` or an autofree variable.
`autofree OTask`, `autofree ^T`, `autofree pointer`	Autofree types; variables release their heap allocations automatically.
Pointer, `rostr`, `strslice`, `[]T`, ordinary object alias	Borrows existing storage; does not keep it alive.

Keep borrowed storage alive for as long as you use it. For example, a slice of a local array cannot be returned for use after that local array has disappeared.

An object variable declared without `autofree` does not destroy the object automatically. Assigning it to another variable does not change that:

```
var task : OTask = new OTask("download")
var another : OTask = task
```

There is one task object, and both variables let you access it. The second declaration does not create a second object, and neither variable has an autofree type. When both variables leave scope, the object remains allocated. DQ does not find and release it for you.

For automatic cleanup, declare `task` with an autofree type. Otherwise, call `delete` when you are finished with the object. After it is destroyed, you must stop using it through either variable; having another variable refer to the object does not keep it alive.

Strings and dynamic arrays release their own storage automatically, but an array's storage is separate from any heap objects its elements refer to. For example, releasing a `[*]OTask` array does not destroy its task objects. Use `[*]autofree OTask` when the array should also destroy those objects during cleanup.

2.16.2 Explicit Allocation and Release

Allocate heap objects with `new` and raw byte buffers with `MemAlloc`. Use `delete` to release either allocation. The allocation stays in memory until you release it; leaving the function does not release it automatically.

This complete program demonstrates both forms. Each function releases its allocation when its work finishes normally. `RunTask` prints the supplied mode, but returns early if the mode is empty. `ProcessData` fills its buffer with `fillvalue`, which must be between 0 and 255 because each byte holds a value in that range. The early-return branches **deliberately contain a mistake**: they return before reaching the cleanup.

```
use print

object OTask:
    name : str

    func *Create(aname : strslice):
        name = aname
    endfunc

    func *Destroy():
        @.PrintLn("Finished: {}", [name])
    endfunc
endobject

func RunTask(amode : rostr):
    var task : OTask = new OTask("download")
    if amode == "":
        PrintLn("Task mode must not be empty")
        return // mistake: skips delete, leaving the task allocated
    endif

    PrintLn("Working on {} in {} mode", [task.name, amode])
    delete task
endfunc

func ProcessData(fillvalue : int):
    var buffer : ^byte = ^byte(MemAlloc(256))
    if buffer == nil:
        PrintLn("Could not allocate buffer")
        return // no allocation was made, so there is nothing to release
    endif
    if fillvalue < 0 or fillvalue > 255:
        PrintLn("Invalid fill value: {}", [fillvalue])
        return // mistake: skips delete, leaving the buffer allocated
    endif

    MemSet(buffer, fillvalue, 256)
    PrintLn("First byte: {}", [buffer[0]^])
    delete buffer
    PrintLn("Buffer released")
endfunc

func *Main():
    PrintLn("-- normal completion")
    RunTask("fast")
    ProcessData(42)

    PrintLn("-- early return")
    RunTask("")
    ProcessData(300)
    PrintLn("Back in Main")
endfunc
```

Output when the allocations succeed:

```
-- normal completion
Working on download in fast mode
Finished: download
First byte: 42
Buffer released
-- early return
Task mode must not be empty
Invalid fill value: 300
Back in Main
```

On the normal path, `delete task` calls the task's destructor and releases its storage. `delete buffer` releases the byte buffer; there is no object destructor to call for it. These variables are not used again before their functions end, so there is no need to clear them. Other variables that refer to the same allocation must no longer be used to access it either. Never delete an embedded object created with `<-`.

When the mode is empty or the fill value is outside 0 to 255, `return` immediately leaves the function. It skips all remaining statements, including `delete`. For example, `ProcessData(300)` allocates its buffer before checking the value, then returns without

freeing it. Notice that the second pair of calls prints neither `Finished: download` nor `Buffer released`. Both functions return to `Main`, but the task object and the 256-byte buffer from those calls are still allocated.

Their local variables disappear when the functions return, so the program no longer has a way to reach those allocations and release them. This is a **memory leak**. Repeating these calls would leave more memory allocated each time, even though no work was completed.

You could add the missing release before each early `return`. However, every additional exit path needs the same care, including exceptions. The next section introduces autofree variables, which release their allocations on both normal completion and early return, without a separate cleanup call on each path.

The implicit runtime also provides `MemFree` for releasing raw storage and `MemFreeRef` for releasing it and clearing a generic `pointer` variable in one call. External resources often have their own close or release function; match each resource to the operation its API requires.

2.16.2.1 Reusing a Variable After `delete`

For a variable without `autofree`, `delete buffer` releases the allocation but leaves the old address in the variable. That address is no longer valid. If the variable stays in scope, use `delete buffer = nil` to release the allocation and set the variable to `nil` in one statement:

```
var buffer : ^byte = ^byte(MemAlloc(256))
// use the buffer

delete buffer = nil // release the storage and mark the variable as empty

// Later, allocate a new buffer if there is none.
if buffer == nil:
    buffer = ^byte(MemAlloc(512))
endif
// use the new buffer

delete buffer // finished with this variable
```

Here, `nil` tells later code that the variable currently refers to no buffer. Without clearing it, `buffer == nil` would not detect that the first allocation has been released. The same applies to an object variable without `autofree`. Clearing one variable does not clear other variables that refer to the released allocation.

You can also assign a new allocation directly after `delete`; setting `nil` first is not required for reassignment. It is useful when later code needs to check whether the variable currently holds an allocation. Assigning `nil` alone to a variable without `autofree` does not release its allocation.

2.16.3 `autofree` Types and Variables

2.16.3.1 Declaring Autofree Variables

Use `autofree` for a heap allocation that should be released automatically. This complete program creates two task objects in `Main` and allocates a structure and a byte buffer in `ProcessData`. The task destructor prints a message so you can see when object cleanup happens:

```
use print

object OTask:
    name : str

    func *Create(aname : strslice):
        name = aname
    endfunc

    func *Destroy():
        @.PrintLn("Finished: {}", [name])
    endfunc
endobject

struct SPoint:
    x : int
    y : int
endstruct

func ProcessData(fillvalue : int):
    var point : autofree ^SPoint = new SPoint
    point.x = 10
```

```
    point.y = 20

    var buffer : autofree ^byte = ^byte(MemAlloc(256))
    if buffer == nil:
        PrintLn("Could not allocate buffer")
        return
    endif
    if fillvalue < 0 or fillvalue > 255:
        PrintLn("Invalid fill value: {}", [fillvalue])
        return
    endif
    MemSet(buffer, fillvalue, 256)

    PrintLn("Point: ({}, {})", [point.x, point.y])
    PrintLn("Buffer filled with {}", [fillvalue])
    PrintLn("Leaving ProcessData")

    // no manual free is required for point and buffer
endfunc

func *Main():
    var first = new autofree OTask("download")
    var second : autofree OTask = new OTask("resize")
    PrintLn("Tasks: {}, {}", [first.name, second.name])

    ProcessData(42)
    PrintLn("Back in Main")
endfunc
```

Output:

```
Tasks: download, resize
Point: (10, 20)
Buffer filled with 42
Leaving ProcessData
Back in Main
Finished: resize
Finished: download
```

`first` uses the `new autofree` shorthand. `second` declares its autofree type explicitly and assigns a newly created object to it. Both forms arrange for the object to be destroyed automatically.

`point` has type `autofree ^SPoint` and releases the structure's heap storage automatically. `buffer` has type `autofree ^byte` and releases its 256-byte allocation automatically. The cast `^byte(...)` gives the raw allocation a byte-pointer type. `MemSet(buffer, fillvalue, 256)` fills all 256 bytes with the requested value.

When `ProcessData` returns, its local autofree variables release their allocations in reverse declaration order: first `buffer`, then `point`. This happens after `Leaving ProcessData` is printed and before `Main` prints `Back in Main`. Releasing these non-object allocations produces no destructor messages; their storage is already freed when execution returns to `Main`. If `fillvalue` is outside 0 to 255, the early `return` also releases both allocations automatically.

The task objects remain available until `Main` ends. Then `second` is destroyed before `first`, producing `Finished: resize` and `Finished: download`. No `delete` or `MemFree` calls are needed in either function.

An autofree variable must refer to a compatible heap allocation. Do not give it a local variable's address, an address inside another allocation, or an external resource handle such as a file handle.

When the initializer already determines the pointed-to type, you can use `autofree ?` to avoid writing that type twice:

```
var buffer : autofree ? = ^byte(MemAlloc(256))
var point : autofree ? = new SPoint
```

The compiler infers `^byte` for `buffer` and `^SPoint` for `point`. The `autofree` type attribute is never inferred: you must write it explicitly. Here, `autofree ?` combines an inferred pointer type with explicitly requested automatic cleanup. The `new autofree` shorthand used for `first` also specifies `autofree` explicitly.

2.16.3.2 Cleanup

An autofree variable releases its current allocation when it is overwritten or leaves its scope. Assigning `nil` releases it early:

```
var task = new autofree OTask("download")
task = new OTask("resize") // destroys download before taking the new object
task = nil                 // destroys resize; task is now empty
```

For objects, cleanup calls `*Destroy` and releases the heap storage. For `autofree ^T` and `autofree pointer`, it releases the allocation without calling an object destructor. An autofree variable set to `nil` has nothing to release.

`delete` also works on an autofree variable and leaves it `nil`:

```
var task = new autofree OTask("download")
delete task // task is now nil; leaving the scope will not delete it again
```

Local autofree variables start as `nil`, and local values are cleaned up in reverse construction order. Managed values, embedded objects, and autofree variables receive cleanup on normal block exit, `return`, loop transfers, and exception propagation. Autofree fields are cleaned up in reverse declaration order when their containing object is destroyed.

Global autofree variables start as `nil`. Replacing them releases their old values, but DQ does not automatically destroy globals at program shutdown. Release global resources explicitly.

2.16.3.3 Borrowing and Moving Ownership

Only one autofree variable should be responsible for a given allocation. Assigning one autofree variable to another **moves ownership**: the destination becomes responsible for releasing the allocation, and the source becomes `nil`:

```
var first = new autofree OTask("download")
var second : autofree OTask = first
// first is now nil; second owns the download task
```

Assigning an autofree variable to an ordinary reference instead **borrows** the object:

```
var task = new autofree OTask("download")
var borrowed : OTask = task
borrowed.name = "resize" // changes the same object; task still handles cleanup
```

The borrow becomes invalid when the autofree variable releases the object. Do not delete the borrowed reference separately or assign it to another autofree variable. DQ does not track every borrowed reference and clear it when the object is destroyed; a non-`nil` borrow can still be invalid.

Moving ownership does not move the object in memory. Existing borrows still refer to that object, but its lifetime now depends on the destination autofree variable. The source becomes `nil`, so check or replace it before using it again.

Source	Assign to `autofree T`	Assign to ordinary `T`
Ordinary `T`	Take ownership; the source remains an alias.	Copy the reference.
`autofree T` variable	Move ownership; set the source to `nil`.	Borrow; the source still handles cleanup.

Replacing an autofree variable first releases its previous allocation. Assigning an autofree variable to itself leaves it unchanged.

2.16.3.4 Autofree Variables and Functions

An ordinary object or pointer parameter borrows its argument, including when the caller passes an autofree variable. The call does not clear that variable or transfer responsibility for cleanup:

```
func Rename(task : OTask, name : strslice):
    task.name = name
endfunc
```

Calling `Rename(task, "resize")` changes the object's name while leaving ownership with the caller.

Use `ref autofree T` when the function needs to replace the caller's autofree variable:

```
func ReplaceTask(task : ref autofree OTask):
    task = new OTask("save") // releases the caller's previous task
endfunc
```

A `ref autofree T` parameter requires an autofree variable; a `ref T` parameter requires ordinary reference storage. Their `autofree` qualifications must match. To return a new object, use an ordinary object result and let the caller assign it to an autofree variable. `autofree` return types are not supported.

2.16.3.4.1 RETURNING A NEW OBJECT

A factory is a function that creates a value for its caller. It can return a new heap object directly:

```
func CreateTask(name : strslice) -> OTask:
    return new OTask(name)
endfunc

func *Main():
    var task : autofree OTask = CreateTask("download")
    Rename(task, "resize")
    ReplaceTask(task)
    PrintLn("Working on {}", [task.name])
endfunc
```

Put this code after `OTask`, `Rename`, and `ReplaceTask` from the examples above. It prints:

```
Finished: resize
Working on save
Finished: save
```

`Rename` borrows the object. `ReplaceTask` changes the caller's autofree variable, destroying the renamed object before storing the new one. The final object is destroyed when `Main` ends.

The ordinary `OTask` result type does not say whether a function returns a new object or borrows an existing one. Document that distinction for callers. Never return a local embedded object, or an object still owned by a local `autofree` variable: local cleanup would destroy it before the caller could use it.

2.16.3.5 Autofree Fields

An object can take responsibility for another object through an autofree field:

```
object OWorker:
    current : autofree OTask
endobject

func *Main():
    var worker <- OWorker()
    var task = new autofree OTask("download")
    worker.current = task // moves ownership into the field; task becomes nil
    PrintLn("Working on {}", [worker.current.name])
endfunc
```

With the earlier `OTask` definition and `use print`, this prints `Working on download` followed by `Finished: download`. Destroying `worker` automatically cleans up `current`; `OWorker` needs no custom destructor. Use an ordinary `OTask` field for a borrowed link, such as a reference back to a containing object. Giving two fields ownership of the same allocation would cause both to attempt to release it.

2.16.3.6 Arrays

An array with an autofree element type automatically releases its objects:

```
var tasks : [*]autofree OTask
tasks.Append(new OTask("download"))
tasks.Append(new OTask("resize"))
tasks.Delete(0) // destroys download; resize remains in the array
```

Replacing or removing an element destroys its previous object. New empty elements are `nil`. Assigning a dynamic array to another variable still shares the same array; its remaining objects are destroyed when the last reference to that array is released.

Single-element `.Append()`, `.Prepend()`, and `.Insert()` take ownership of an ordinary reference or move a source of an autofree type. `.Pop()` and `.PopFirst()` remove an object without destroying it and return an ordinary reference. Assign the result to an autofree variable or delete it explicitly:

```
var tasks : [*]autofree OTask
tasks.Append(new OTask("download"))
var next : autofree OTask = tasks.Pop() // next now owns the removed task
```

Use `for item : ref in tasks` to replace array elements in place. The reference keeps the element's autofree type, so replacement frees the previous object. See Dynamic Array `autofree` Elements for more examples.

2.16.3.6.1 ARRAY COPYING RESTRICTIONS

Assigning a fixed array of autofree elements moves every element: each destination's previous value is released, and each source element becomes `nil`. Self-assignment leaves the array unchanged.

Dynamic arrays of autofree elements cannot use `.Clone()` or copy a range with `.Append()`, `.Prepend()`, `.Insert()`, or the slice variants. Copying those objects into separate autofree elements would duplicate ownership. Borrowed `[]autofree T` views can still be indexed, iterated, and passed to read-only APIs. Single-element insertion can move an element from the same array even when the operation moves the array's storage.

2.16.3.7 Placement Restrictions and Type Inference

`autofree` can qualify an object type, a typed pointer, or `pointer`. It can be used for variables, object fields, fixed- or dynamic-array elements, and `ref` parameters. It cannot be used directly on numbers, booleans, characters, structures, unions, function references, or whole array types.

It is also not supported on embedded `<-` fields, structure or union fields, properties or their indices, function results, or `refin` and `refout` parameters. Type aliases may name autofree types, with the same placement rules.

Type inference never adds the `autofree` attribute. Use `autofree ?` to infer the underlying type while specifying automatic cleanup explicitly. Bare `:?` produces a type without `autofree`, even when the source is an autofree variable. For the `new autofree T` declaration shorthand, omit `:?` as in the examples above. The same distinction applies to inferred `for ... in` variables.

2.16.3.7.1 INFERENCE EXAMPLE

```
var task = new autofree OTask("download")
var borrowed :? = task           // inferred OTask; task is unchanged
var next : autofree ? = task     // inferred OTask, explicit autofree; task becomes nil
```

`borrowed` still refers to the same object, now owned by `next`. Prefer an explicit autofree type at a transfer point if it makes the responsibility easier to see.

2.17. Exceptions and Error Handling

An **exception** reports a failure, such as invalid input or an operation that cannot be completed. It stops the current work and looks for code that handles the problem. Use `try` and `except` to handle a failure without ending the whole program.

Custom exception type names conventionally start with a capital `E`, as in `EInput`.

```
use print

object EInput(Exception):
endobject

func RequirePositive(value : int):
    if value <= 0:
        raise EInput("expected a positive value, got {}", [value])
    endif
endfunc

func *Main():
    try:
        RequirePositive(0)
        PrintLn("Value accepted")
    except EInput as error:
        PrintLn("Error: {}", [error.message])
    endtry
    PrintLn("Ready for another value")
endfunc
```

This prints:

```
Error: expected a positive value, got 0
Ready for another value
```

`raise` reports the problem. Execution skips the remaining statements in the `try` block and enters the matching `except` handler. `as error` gives the handler a name for the exception, so it can read the message.

2.17.1 Three Ways to Handle File Errors

Consider a function that logs three measurements: temperature, relative humidity, and pressure. Each version opens `measurements.log` in append mode and makes three separate writes, one per measurement. Each line includes the same Unix timestamp, in seconds, obtained from `Time()`.

2.17.1.1 No Error Handling

This version calls libc directly and ignores all return values:

```
use time

func libc_fopen(filename : ^char, mode : ^char) -> pointer [[external('fopen')]] ..
func libc_fprintf(stream : pointer, fmt : ^char, ...) -> int32 [[external('fprintf')]] ..
func libc_fclose(stream : pointer) -> int32 [[external('fclose')]] ..

func LogMeasurements(temp : float, rh : float, pressure : float):
    var timestamp : float64 = Time()
    var f : pointer = libc_fopen("measurements.log", "a")
    libc_fprintf(f, "%.3f temp=%.2f\n", timestamp, temp)
    libc_fprintf(f, "%.3f rh=%.2f\n", timestamp, rh)
    libc_fprintf(f, "%.3f pressure=%.2f\n", timestamp, pressure)
    libc_fclose(f)
endfunc
```

The code is short, but it cannot tell the caller whether logging succeeded. If opening fails, `f` is `nil`, and passing it to `libc_fprintf` is invalid and may crash the program. If a write fails, the function still attempts the remaining writes and silently loses data.

2.17.1.2 Return-Value-Based Error Handling

With the same libc declarations, the function can check each result and return `false` on failure. `libc_fopen` returns `nil` on failure; `libc_fprintf` returns a negative value on failure. A successful open does not guarantee that any of the writes will succeed, so each write needs its own check.

```
use time
use print

func libc_fopen(filename : ^char, mode : ^char) -> pointer [[external('fopen')]] ..
func libc_fprintf(stream : pointer, fmt : ^char, ...) -> int32 [[external('fprintf')]] ..
func libc_fclose(stream : pointer) -> int32 [[external('fclose')]] ..

func LogMeasurements(temp : float, rh : float, pressure : float) -> bool:
    var timestamp : float64 = Time()
    var f : pointer = libc_fopen("measurements.log", "a")
    if f == nil:
        return false
    endif
    if libc_fprintf(f, "%.3f temp=%.2f\n", timestamp, temp) < 0:
        libc_fclose(f)
        return false
    endif
    if libc_fprintf(f, "%.3f rh=%.2f\n", timestamp, rh) < 0:
        libc_fclose(f)
        return false
    endif
    if libc_fprintf(f, "%.3f pressure=%.2f\n", timestamp, pressure) < 0:
        libc_fclose(f)
        return false
    endif
    return libc_fclose(f) == 0
endfunc

func *Main():
    if not LogMeasurements(21.5, 48, 1013.2):
        PrintLn("Could not log measurements")
    endif
endfunc
```

Every failure path after opening closes the file before returning. Closing also needs a check on the success path: libc buffers output, so a failure may only be reported when `libc_fclose` flushes the remaining data. After a write has already failed, the function returns `false` regardless of the close result. The caller must check the function's return value too; intermediate callers would need to pass that failure on explicitly.

2.17.1.3 Exception-Based Error Handling

The standard `file` package's `OFile` object raises `EFileError` when opening or writing fails. The logging function can therefore express the sequence of operations directly:

```
use file
use time
use print

func LogMeasurements(temp : float, rh : float, pressure : float):
    var timestamp : float64 = Time()
    var f <- OFile()
    f.Open("measurements.log", "a")
    f.WriteText(Format("{:.3f} temp={:.2f}\n", [timestamp, temp]))
    f.WriteText(Format("{:.3f} rh={:.2f}\n", [timestamp, rh]))
    f.WriteText(Format("{:.3f} pressure={:.2f}\n", [timestamp, pressure]))
endfunc

func *Main():
    try:
        LogMeasurements(21.5, 48, 1013.2)
    except EFileError as error:
        PrintLn("Could not log measurements: {}", [error.message])
    endtry
endfunc
```

Each `WriteText` checks for failure inside `OFile`. If opening or any write raises an exception, the remaining operations are skipped and the exception travels to the caller's handler. There is no error result to check or pass on in `LogMeasurements`.

The local `f` uses autofree ownership (`<-`), so its destructor closes the file both on normal return and during exception propagation. Currently, `OFile.Close()` does not report libc close errors, so this version cannot detect failures reported only when buffered output is flushed at close time.

None of these approaches makes the three writes atomic: if a later write fails, earlier lines may already have been written. Handling an error does not undo those writes.

2.17.2 Advantages of the Exception-Based Error Handling

The logging examples show several advantages of exceptions:

- **The normal sequence stays short and readable.** Opening the file and writing the three measurements appear together, without repeated checks and early returns between them. Adding another write does not require another error branch in the logging function.
- **Error details travel with the failure.** The boolean result only says that logging failed. An exception carries a type and **message**, and its **backtrace** helps locate the failure. Callers can distinguish error types without adding separate output parameters to every intermediate function.
- **Errors cannot be silently ignored.** With return values, forgetting a check can let the program continue working with invalid or incomplete data. An exception automatically stops the failed operation, even when the programmer has not supplied an error handler. The program normally reports the unhandled exception with its message and backtrace, then terminates when no matching error handler provided.
- **Cleanup code does not need to be repeated.** The common cleanup block - `except` or `finally` - makes the source code more maintainable and readable.

2.17.3 Raising and Matching Exceptions

`Exception` is supplied by the implicit runtime when exception support is enabled. Its constructor accepts a message, optionally with formatting arguments, just like `PrintLn`.

You can define your own exception types to distinguish failures. This example uses libc's `scanf` to read a number from the console. It raises `Exception` if no number can be read, or `EInput` if the number is greater than 100:

```
use print

func libc_scanf(fmt : ^char, ...) -> int32 [[external('scanf')]] ..

object EInput(Exception):
endobject

func *Main():
    try:
        PrintLn("Enter a number no greater than 100:")
        var value : float64 = 0
        if libc_scanf("%lf", %value) <> 1:
            raise Exception("Could not read a number")
        endif
        if value > 100:
            raise EInput("Number is too big: {}", [value])
        endif
        PrintLn("Number accepted: {}", [value])
    except EInput as error:
        PrintLn("Input problem: {}", [error.message])
    except Exception as error:
        PrintLn("Other problem: {}", [error.message])
    endtry
endfunc
```

For example, entering `42` prints `Number accepted: 42`, entering `150` prints `Input problem: Number is too big: 150`, and entering `hello` prints `Other problem: Could not read a number`.

The `%lf` conversion stores a `float64` at the address `%value`. `scanf` returns 1 when it reads that number successfully. It reads a numeric prefix, so input such as `42abc` also reads as 42; this example does not validate the entire line.

A handler for `Exception` also matches exceptions derived from it, including `EInput`. Placing the `EInput` handler first lets it handle the too-big case separately.

If a handler does not need the exception value, omit `as error`:

```
except EInput:
    // handle an input problem without reading its message
```

If no handler matches, the exception continues to the caller. Do not call `delete` on the handler's exception variable; the exception machinery manages that object.

2.17.3.1 Re-raising an Exception

A handler can report a failure or do some local work, then let its caller handle the same failure. Use `raise` without an argument inside the handler to re-raise the current exception:

```
use print

object EInput(Exception):
endobject

func RequirePositive(value : int):
    if value <= 0:
        raise EInput("Expected a positive value, got {}", [value])
    endif
endfunc

func ProcessValue(value : int):
    try:
        RequirePositive(value)
        PrintLn("Processing {}", [value])
    except EInput as error:
        PrintLn("Could not process value: {}", [error.message])
        raise
    endtry
endfunc

func *Main():
    try:
        ProcessValue(0)
    except EInput as error:
        PrintLn("Please try again: {}", [error.message])
    endtry
endfunc
```

Output:

```
Could not process value: Expected a positive value, got 0
Please try again: Expected a positive value, got 0
```

`ProcessValue` reports the problem, but its bare `raise` passes the exception on to the handler in `Main`. The `Processing` line is skipped. Re-raising preserves the original exception type, message, and backtrace, so the failure still points to `RequirePositive`. If no outer handler catches it, the exception reaches the default error handler.

2.17.4 Cleanup with `finally`

A `finally` block performs an action whenever execution leaves a `try` block. It runs after normal completion or a handled error, and also when an exception continues to the caller:

```
use print

func *Main():
    try:
        PrintLn("Starting work")
        raise Exception("Connection lost")
    except Exception as error:
        PrintLn("Error: {}", [error.message])
    finally:
        PrintLn("Work finished")
    endtry
endfunc
```

This prints `Starting work`, `Error: Connection lost`, and `Work finished`. You can also use `try` with `finally` alone when you want cleanup without handling the exception there.

`finally` also runs when `return`, `break`, or `continue` leaves the protected block. It is useful for an explicit action such as restoring a setting or releasing an external resource. For objects owned by your code, prefer automatic cleanup: local autofree variables, managed values, and embedded objects already clean themselves up during exception propagation.

2.17.4.1 Restoring State Without Handling the Error

A `finally` block is useful even when cleanup does not involve memory. This complete program temporarily changes a setting and restores it before a caller handles the error:

```
use print

var quiet : bool = false

func RunQuietly():
    var previous : bool = quiet
    quiet = true
    try:
        raise Exception("work failed")
    finally:
        quiet = previous
        PrintLn("Setting restored")
    endtry
endfunc

func *Main():
    try:
        RunQuietly()
    except Exception as error:
        PrintLn("Error: {}; quiet={}", [error.message, quiet])
    endtry
endfunc
```

Output:

```
Setting restored
Error: work failed; quiet=false
```

`RunQuietly` has no `except` handler, so the exception continues to `Main`. Its `finally` block runs first. Saving the previous setting also makes this work when `quiet` was already `true` before the call.

2.17.5 Messages and Backtraces

The exception's `.message` describes the problem. A **backtrace** shows the sequence of calls that led to it. If no handler catches the exception, the default error handler prints both and ends the program with a failure status.

Save this example as `backtrace.dq`. It calls through several functions and lets the exception reach the default error handler:

```
func CheckTemperature(temp : float):
    if temp > 100:
        raise Exception("Temperature is too high: {}", [temp])
    endif
endfunc

func LogTemperature(temp : float):
    CheckTemperature(temp)
endfunc

func ReadSensor():
    LogTemperature(150)
endfunc

func *Main():
    ReadSensor()
endfunc
```

Run it with `dq-run backtrace.dq`. Example output on Linux, with the source paths shortened to the filename:

```
UNCAUGHT EXCEPTION: Exception: Temperature is too high: 150
Backtrace =
  #0 CheckTemperature @ backtrace.dq:3
  #1 LogTemperature @ backtrace.dq:9
  #2 ReadSensor @ backtrace.dq:13
  #3 Main @ backtrace.dq:17
```

Read the backtrace from the failure outward: `CheckTemperature` raised the exception, called by `LogTemperature`, which was called by `ReadSensor`, which was called by `Main`. The source locations help you find each step in the code.

To get source code line info at the backtraces the executable must be compiled with `-g`. Code optimization might "scramble" the source code, so for clean source code info the optimization might need to be turned off with `-O0`. The `dq-run` uses `-g -O0` by default to provide best error reporting. The `dq-comp` however uses `-O1` by default for better performance.

The backtrace info is available on Windows and Linux, but might not be available on every target.

2.17.6 Cleanup Details and Limitations

Initialized local values are cleaned up in reverse construction order as an exception leaves their scope. If a constructor fails, already initialized autofree fields and base state are also cleaned up.

Catching an exception does not undo changes made before the failure. Restore or discard partially updated data as appropriate for the operation. If a `finally` block raises another exception, the new exception becomes the active failure. Calling an external process-exit function does not perform normal block-exit cleanup.

2.17.7 Zero-Cost Exception Handling

DQ uses **zero-cost exception handling** based on unwind tables on supported hosted targets. The compiler records which parts of a function are protected and where their cleanup code and exception handlers are located. These tables are stored in the executable; they do not need to be checked after each call.

Function calls inside a `try` block remain ordinary calls in the generated machine code. They are not wrapped in an extra error-checking function, and there is no hidden exception-result check after every call. When no exception is raised, execution follows the normal sequence of operations.

When an exception is raised, the runtime uses the unwind tables to walk back through the active function calls, run the required cleanup, and transfer control to a matching handler. This work happens on the error path.

“Zero-cost” refers to the absence of per-call exception checks during normal execution. It does not mean that raising an exception is free: finding a handler, capturing a backtrace, and running cleanup take time. The tables and handler code also increase executable size, and exception handling can affect compiler optimizations. Ordinary cleanup, including `finally` blocks, still runs wherever the program requires it.

2.17.8 Runtime Errors and Target Support

Runtime checks, such as range and encoding checks, can also report failures. Some targets convert hardware faults into exceptions: Linux runtimes and Windows x86-64 have fault handling, with recovery verified at `-O0`. Optimized fault recovery has backend/runtime limitations. Validate addresses before use; fault handling is not a portable way to make invalid memory access safe.

Exceptions normally default to enabled on hosted targets and disabled on bare targets. The compiler options `--exceptions` and `--no-exceptions`, or a project setting, select support. `EXCEPTIONS` is defined when that support is enabled. WASI does not support DQ exceptions.

Runtime checks still apply when exception handling is disabled, but failures cannot be recovered with DQ handlers in that configuration. See Compiler Directives and Runtime and Program Execution.

2.18. anyvalue

`anyvalue` holds one value that can be a number, text, a boolean, or another supported basic type. It remembers which kind of value it holds. Use it when a variable or collection needs to store values of different kinds.

```
use print

func *Main():
    var value : anyvalue = 42
    PrintLn("{}", [value.AsInt(0)])

    value = "hello"
    PrintLn("{}", [value.AsStr("")])
endfunc
```

This prints `42` and then `hello`. The same variable first holds an integer, then text. The accessor methods, such as `AsInt`, let you read a value in the form you need. Unlike an ordinary `int` or `str`, an `anyvalue` does not have one fixed kind of content throughout its lifetime. Use a specific type when the kind is known in advance; use `anyvalue` at interfaces that need mixed values.

2.18.1 Checked Access

Use an `Is...` method to check the kind of value. Use an `As...` method to read it, supplying a default for values that cannot be converted:

```
var value : anyvalue = "hello"
var number : int = value.AsInt(-1) // -1: text is not converted to a number
var text : str = value.AsStr("")  // "hello"
```

The default is also a possible real value. If you need to distinguish a stored `-1` from a failed conversion, check the kind first:

```
if value.IsInt():
    var number : int = value.AsInt(0)
    // use the stored integer
endif
```

Some accessors accept more than one related kind; for example, `AsFloat` can read an integer as a floating-point value. They do not perform arbitrary conversions such as parsing text into numbers.

2.18.2 Array Parameters

A `[]anyvalue` parameter accepts a list containing different kinds of values. Read each element with an accessor before passing it to a formatter, which accepts `varargs`.

```
use print

func LogItems(values : []anyvalue):
    PrintLn("{}: {} items, ready={}", [values[0].AsText(""), values[1].AsInt(0), values[2].AsBool(false)])
endfunc

func *Main():
    LogItems(["Shopping", 3, true])
endfunc
```

This prints `Shopping: 3 items, ready=true`. The caller passes a temporary array whose elements are converted to `anyvalue`. Its slice is valid for the duration of the call.

2.18.3 Null

`anyvalue` can also represent **null**, meaning a missing value. Use `SetNull` to set it and `IsNull` to test it:

```
var reading : anyvalue = 21.5
reading.SetNull()
var missing : bool = reading.IsNull() // true
```

Null is different from zero, `false`, an empty string, and a nil pointer.

2.18.4 Supported Values

An `anyvalue` can hold these basic values:

Values	Notes
Signed and unsigned integers	Integer width and signedness are recorded.
Floating-point numbers	`float32`, `float64`, and `float`.
Booleans	`true` or `false`.
Characters	`char` and `wchar`; `char16` is stored as `wchar`.
Text	`str`, `rostr`, `strslice`, and `embstr`.
Pointers	Typed pointers and generic `pointer` values.
Null	A missing value, set with `SetNull()`.

Objects, structures, function references, and arbitrary arrays cannot be stored as general values inside it. The temporary array in a `[]anyvalue` call works because each supported element is converted separately.

2.18.5 Methods

Use this table to choose a check or accessor. `def` means the default value to return when the requested conversion is unavailable.

Method	Meaning
`IsNull()` / `SetNull()`	test or set the null value
`IsNumber()`	true for integer or floating point values
`IsInt()` / `IsSInt()` / `IsUint()`	test integer kind
`AsInt(def)` / `AsUint(def)`	convert integer, float, or character code, otherwise return default
`IsChar()` / `AsChar(def)` / `SetChar(value)`	test, read, or set a byte character
`IsWChar()` / `AsWChar(def)` / `SetWChar(value)`	test, read, or set a Unicode character; setter also accepts `char16`
`IsBool()` / `AsBool(def)`	test/read boolean
`IsPointer()` / `AsPointer(def)`	test/read pointer
`IsFloat()` / `IsFloat32()` / `IsFloat64()`	test floating point kind
`AsFloat(def)` / `AsFloat32(def)` / `AsFloat64(def)`	convert float or integer
`IsText()`	true for `str`, `rostr`, `strslice`, or `embstr` values
`IsStr()`	true only for dynamic `str`
`AsStr(def)`	return a dynamic string
`AsText(def)`	return a text view-style value

A helper can use these methods to produce a label for different kinds of data:

```
use print

func ToLabel(value : anyvalue) -> str:
    if value.IsNull():
        return "<missing>"
    elif value.IsText():
        return value.AsStr("")
    elif value.IsNumber():
        return Format("{}", [value])
    else:
```

```
        return "<value>"
    endif
endfunc
```

Add this entry point after `ToLabel` to try each branch:

```
func *Main():
    var value : anyvalue = 42
    PrintLn("{}", [ToLabel(value)])
    value = "hello"
    PrintLn("{}", [ToLabel(value)])
    value.SetNull()
    PrintLn("{}", [ToLabel(value)])
    value = true
    PrintLn("{}", [ToLabel(value)])
endfunc
```

This prints `42`, `hello`, `<missing>`, and `<value>`, each on its own line. `ToLabel` checks the stored kind before choosing how to represent it. The final branch is this function's policy for values such as booleans, rather than a limitation of DQ's formatting functions.

`Format` is available through the implicit `sys` runtime interface. See String Functions.

2.18.6 Setters and Text Views

In addition to assignment, explicit setters choose what kind of value to store:

Methods	Stored category
`SetInt(value)`, `SetUInt(value)`	Signed/unsigned integer.
`SetBool(value)`, `SetPointer(value)`	Boolean or raw address.
`SetFloat(value)`, `SetFloat32(value)`, `SetFloat64(value)`	Floating-point value.
`SetChar(value)`, `SetWChar(value)`	Byte character or Unicode scalar.
`SetText(value)`, `SetEmbStr(value)`	Borrowed text representation.
`SetStr(value)`	Managed dynamic string.
`SetNull()`	Missing value (null).

`AsStrSlice(default)` is the text-view alias of `AsText(default)`. A borrowed text view returned by an accessor still depends on the stored/source value's lifetime. `SetStr`, `AsStr`, and `IsStr` require dynamic-string support. Method names are case-sensitive; in particular the unsigned setter is `SetUInt`, while its predicate and accessor are `IsUint` and `AsUint`.

2.18.7 Lifetime

Storing a managed `str` keeps its text alive. Storing borrowed text or a pointer does not keep the source alive. For example, a slice into a local buffer must not be used after that buffer has disappeared, even if stored in an `anyvalue`.

To retain text independently of a borrowed source, copy it into managed string storage before keeping it:

```
var buffer : embstr(32) = "ready"
var saved : anyvalue
saved.SetStr(buffer) // stores a managed string copy of the buffer's text
buffer = "changed"
// saved.AsStr("") is still "ready"
```

A view returned by `AsText` or `AsStrSlice` is also borrowed. Keep its source valid for as long as you use the view. Copying and destroying an `anyvalue` automatically performs the appropriate storage handling for its current kind.

2.18.8 Character Details

Characters have their own kind: `IsNumber` and `IsInt` return `false` for them. `AsWChar` can widen a stored `char`; `AsChar` returns its default for a stored `wchar`.

Formatting with `{}`, `{:?}`, or `{:s}` writes a `char` byte directly and encodes a `wchar` as UTF-8. The formats `{:d}`, `{:u}`, `{:x}`, and `{:X}` instead write the character code:

```
PrintLn('wchar="{}"', [wchar(0x20AC)]) // wchar="€"
```

An invalid Unicode scalar, including an isolated `char16` surrogate, produces `\uFFFD` and sets `DQFMT_ARG_ERROR`.

2.18.9 Runtime Type Information

Internally, each value records a broad kind, a more specific representation, and its data. Use the public predicates and accessors rather than depending on the private memory layout. The methods are backed by the runtime's `AnyVal...` functions.

2.19. Modules and Packages

A **module** is a `.dq` source file. Split a program into modules to keep related code together and reuse it from other files. A **package** groups modules under a shared name, such as `json` or `file`.

Here is a complete two-file program. Put both files in the same directory.

`math_extra.dq`:

```
func Add(left : int, right : int) -> int ..

implementation

func Add(left : int, right : int) -> int:
    return left + right
endfunc
```

`main.dq`:

```
use print
use ./math_extra as math --

func *Main():
    PrintLn("{}", [@math.Add(2, 3)])
endfunc
```

Run `dq-run main.dq` to compile both files and print `5`. `use` imports the module, `as math` gives it a short name, and `@math.Add` calls its public function. The trailing `--` keeps that module's names qualified.

The import connects two separate scopes. `Add` belongs to `math_extra.dq`; `Main` belongs to `main.dq`. The alias `math` is the name the importing file uses to reach the other module, not a rename of its source file.

2.19.1 Module Paths from Constants

A `use` path can come from a compile-time string expression enclosed in braces. Any constants in the expression must be visible at that point:

```
use hwconfig
use {UART_MODULE} reexport
```

For example, the project-selected `hwconfig` package can declare:

```
const UART_MODULE : rostr = 'hwimpl/uart_a'
```

Expressions can appear anywhere in the path, including within a path segment. Each expression is evaluated as a constant string and inserted into the path before the complete result is validated:

```
const HWIMPL_ROOT : rostr = 'vihal/arma/rk/impl'
const VARIANT : rostr = 'rk'
use {HWIMPL_ROOT}/rk_utils
use vihal/arma/{VARIANT}/impl/rk_utils as utils --
use vihal/arma/rk/impl/hwuart_{VARIANT} as uart --
```

The expanded path must be a valid module path, without a `.dq` extension. Literal text and expression results are concatenated exactly; no separators are added or removed. Inserted text is not expanded again. Normal `use` modifiers, including `as`, `only`, `exclude`, and `reexport`, apply. Relative paths resolve from the importing module, not the constant's declaration. Runtime strings cannot select modules.

`DQ.PKGNAME` is a read-only compile-time property containing the configured package name of the source file containing the expression, without a trailing slash. Inside an included fragment it uses that fragment's package, just like `^/` in `use` and `#include`. A source outside configured packages produces an error.

Use it to construct a path that retains its meaning when consumed by another package:

```
const HWUART_IMPL : rostr = "\{DQ.PKGNAME}/arma/rk/impl/hwuart_rk"
use {HWUART_IMPL}
```

String concatenation with constant operands (including `"a" + "b"` and named string constants) is folded at compile time, even with dynamic strings disabled.

The configuration module and selected implementation are ordinary dependencies. Changing the constant or resolving the configuration package from a different source invalidates affected cached interfaces. Use different build tags for concurrent configurations sharing a package build root.

2.19.2 Public and Private Declarations

Declarations before `implementation` are public: other modules can use them. Declarations after it are private, except for definitions of functions already declared in the public section.

Use this split when callers need only the public signature:

```
const PUBLIC_VALUE : int = 1
func PublicValue() -> int ..

implementation

const PRIVATE_VALUE : int = 2

func PublicValue() -> int:
    return PUBLIC_VALUE + PRIVATE_VALUE
endfunc
```

An importing module can use `PUBLIC_VALUE` and `PublicValue`, but cannot access `PRIVATE_VALUE`. A public function can also have its body directly in the public section; `implementation` is optional.

Public `constexpr` declarations are available in importing modules. The compiler expands their expressions using the arguments supplied by the importer and the names resolved in the defining module.

2.19.3 Basic Imports

Use a package name to import an installed module, or a relative path for a nearby source file. Omit the `.dq` extension:

```
use print
use ./helpers
```

By default, public names are available directly and through the module's namespace. For example, after `use print`, both `PrintLn(...)` and `@print.PrintLn(...)` are available. A namespace keeps names from different modules separate.

The compiler normally supplies the `sys` runtime interface automatically, so basic runtime functions do not need a separate `use sys`.

2.19.4 Aliases and Merge Control

`as` chooses a namespace name. Add `--` to require that name at each use:

```
use ./math_extra as math --
var total : int = @math.Add(2, 3)
```

This helps when two modules have functions with the same name. You can instead choose which public names are available without qualification:

Import form	Names available directly
`use ./helpers`	All eligible public names.
`use ./helpers --`	None; use `@helpers.Name`.
`use ./helpers only(Read, Write)`	Only `Read` and `Write`.
`use ./helpers exclude(Debug)`	All eligible public names except `Debug`.

`--`, `only`, and `exclude` are alternatives; do not combine them on one import. Qualification through the namespace remains available. For example, `only(Read)` makes `Read` available by its short name, but does not make other public declarations private: they can still be reached as `@helpers.Name`. Use `implementation` in the defining module to hide a declaration from callers.

2.19.5 Relative and Root Paths

Use these forms to find files within your project:

Path	Starting point
`use ./child`	The importing file's directory.
`use ../sibling`	Its parent directory.
`use ^/root_child`	The containing package root (packages only).

Relative imports may contain any number of `../` components, including paths outside the original module or package root. A missing destination produces `ModuleNotFound`.

For example, consider this project layout:

```
project/
    main.dq
    helpers.dq
    tools/
        report.dq
```

From `main.dq`, `use ./helpers` imports `helpers.dq`, and `use ./tools/report` imports `report.dq`. From `tools/report.dq`, `use ../helpers` reaches the same helper module, including when `report.dq` is compiled or edited directly. No source-root option is needed for parent traversal.

`^/` imports require membership in a configured package. The compiler and language server resolve source and package paths through symlinks and search package roots in reverse configuration order. Symlinked package directories are supported; a shadowed package copy does not acquire the active package identity. Standalone files outside configured packages cannot use `^/`.

Bare imports such as `use json` search for installed packages. The root module conventionally has the package name, for example `json/json.dq`. `use nanonet/nano_http` selects another module inside a package.

2.19.6 Package Search

A package is a source directory containing a module tree. Build metadata is optional; it does not replace the source modules.

The compiler searches standard and installation directories, your user package directory, and directories supplied with `--pkg_path`. An explicit package path can override an installed package. If several `--pkg_path` options are given, the last takes precedence.

An explicit mapping such as `--pkg=board:boards/adcif` binds a logical package name to a directory directly, without an extra `board/` directory level. Mappings take priority over every search root. Command-line mappings override project mappings; within each level the last mapping for a name wins. In a project file, write `pkg = 'board:../boards/adcif'`. See package lookup for path anchoring and validation rules.

These same package search paths configure include discovery for `#include "package/file.dqi"`; projects do not need a separate include-directory list. The project-file `pkg_path` setting likewise applies to both module imports and package-qualified includes. Explicit `pkg` mappings apply to both as well. The old `--pkg-path` and `packagepath` spellings remain supported aliases.

2.19.7 Namespace and Symbol Conflicts

If two imported modules expose the same name, qualify it or restrict which names they make available. Import order does not silently choose one meaning. Conflicting local declarations, aliases, selected names, and reexports are also checked.

`@.` explicitly selects the current module's global scope. Inside an object method, use it to call a module function rather than an object member.

2.19.8 Initialization

A module can define `*ModuleInit()` for setup that must happen before `*Main`:

```
use print

var greeting : str

func *ModuleInit():
    greeting = "Ready"
endfunc

func *Main():
    PrintLn("{}", [greeting])
endfunc
```

This prints `Ready`. A used module initializes before the module that uses it, and each module initializes once. Global embedded objects are also initialized before the application starts.

There is no automatic module/global finalization phase. Explicitly release long-lived resources when the application shuts down.

2.19.9 Reexports

`reexport` makes an imported module's public names available through your own module's public interface:

```
use ./leaf only(Value, Run) reexport
```

Use this to provide one convenient entry point for a group of modules. Selection and alias rules apply before reexporting, and conflicting names are reported as errors. Put a reexport in the public section, before `implementation`.

For example, a library can keep its work in `reader.dq` and `writer.dq`, and provide a single `storage.dq` entry point:

```
// storage.dq
use ./reader only(Read) reexport
use ./writer only(Write) reexport
```

A caller can then write `use ./storage` and use `Read` and `Write` without importing the two implementation modules separately. The functions still belong to their original modules; reexporting does not copy their code or run their initialization again. This fragment assumes the two companion modules provide those public functions.

2.19.10 Method-Local Use

Inside an object method, outside variables, constants, and functions (including those from the runtime `sys` module and imported modules) are not directly accessible by their bare names. Placing a method-local `use` statement at the start of a method body allows direct, unqualified access to outside symbols—such as `use socket, sys`, `use .`, or `use *`.

Because this feature specifically controls name resolution inside object methods, its full description, rules, and practical examples are documented in Objects and Properties.

2.19.11 Interface Loading and Cycles

Modules may refer to each other only when the compiler can resolve their public interfaces. A circular dependency that needs unavailable implementation details is an error; the diagnostic shows the dependency chain.

The compiler saves public information in `.dqm_if` files and compiled code in `.o` files. These are generated build artifacts, not files to import in source.

2.19.12 Package Roots and Module Identity

The compiler identifies each module by its package or module root and relative path. Importing the same module through another alias or reexport does not create another runtime instance. Two different source files cannot claim the same module identity with different contents. Repeated imports must also use consistent, unambiguous aliases and selections.

`.dqbuild`, `.o`, and `.dqm_if` files are generated caches and may change between compiler versions. For cache locations and rebuilding, see Build Process and Module Artifacts. The low-level `--no-use-sys` option disables the normally implicit runtime import.

2.19.13 Built-in Search Roots on Linux

The compiler adds these package search roots automatically. `<compiler-dir>` is the directory containing the `dq-comp` executable:

```
/usr/lib/dq/stdpkg
<compiler-dir>/../lib/dq/stdpkg
<compiler-dir>/../stdpkg
/usr/lib/dq/packages
<compiler-dir>/../lib/dq/packages
$HOME/.dq/packages
```

Package roots are checked from last to first. Therefore, a matching package in `$HOME/.dq/packages` takes precedence over the installation roots, and a root provided with `--pkg_path` takes precedence over all built-in roots. If several `--pkg_path` options are given, the last one takes precedence.

2.20. Attributes

An **attribute** gives the compiler extra instructions about a declaration. You have already seen examples such as `[[overload]]` for functions with the same name and `[[virtual]]` for object methods. Other attributes control optimization, memory layout, or access to external code. Most ordinary DQ code needs no attributes; add one when a declaration needs a specific behavior.

Write attributes inside `[[...]]` next to the declaration they affect:

```
use print

func Double(value : int) -> int [[inline]]:
    return value * 2
endfunc

func *Main():
    PrintLn("{}", [Double(21)])
endfunc
```

This prints `42`. `[[inline]]` asks the compiler to consider putting the function's code directly at its call sites. It does not change how you call `Double` or the values it accepts.

2.20.1 Writing Attributes

An attribute belongs to the declaration beside it; it does not set a mode for the rest of the file. Function attributes go after the parameter list and optional result type, before `:` for a body or `..` for a declaration. An attribute can also go before a declaration or in another supported trailing position, as the field and global examples below show.

Some attributes take arguments, such as the symbol name in `[[external('puts')]]`. Combine attributes in one list, separated by commas:

```
func DefaultHandler() [[weak, cexport]]:
    // fallback implementation
endfunc
```

2.20.2 Attribute Summary

Not every attribute applies to every declaration. Use the table to find the attribute for your task; less common details follow the examples.

Attribute	Purpose and applicable declarations
`external`, `external('name')`	Bind a function or global variable to a native symbol.
`external('name', 'module')`	Select a WebAssembly function import module.
`export('name')`	Export a function or global value with an explicit native name.
`cexport`	Use the declaration name as its unmangled native export name.
`used`	Retain a function or global value whose references are invisible to the optimizer.
`section('name')`	Place a function or global value in a named native section.
`align(n)`	Request alignment for supported global storage or compound fields.
`packed`	Pack a structure/compound layout.
`overload`	Mark a function, method, or constructor as an overload.
`inline`, `always_inline`, `noinline`	Control inlining of functions.
`virtual`, `override`, `abstract`, `final`	Control object method dispatch and inheritance.
`asm`	Supply a function body in target assembly.
`weak`	Give a module-level function definition replaceable weak linkage.
`volatile`, `noread`, `nowrite`	Control direct low-level global/field access.
`regrw`, `regro`, `regwo`	Volatile register read/write, read-only, and write-only shorthands.
`nowarn`	Suppress warnings on supported declarations/statements.

2.20.3 Function Inlining

Inlining can avoid the cost of a function call, but can also increase program size. `[[inline]]` is a hint; `[[always_inline]]` makes a stronger request, and `[[noinline]]` prevents inlining. Normal `[[inline]]` is considered at `-O1` and above.

Inlining across module boundaries requires full link-time optimization (`--lto=full`), which lets the optimizer see function bodies from other modules.

2.20.3.1 Optimization Levels

At `-O1`, only functions marked `inline` or `always_inline` are eligible for inlining. Unmarked definitions receive LLVM's `noinline` attribute, including in full LTO builds. An explicit `noinline` still prevents inlining of a function marked `inline`. Higher optimization levels can also inline unmarked functions.

2.20.4 Linkage and Placement

Use `[[external]]` for a function or global variable supplied by native code. Use `[[cexport]]` to expose a DQ declaration under its own name, or `[[export('native_name')]]` to choose another name:

```
var counter : uint32 = 0 [[export('dq_counter'), align(4), used]]

func ReadCounter() -> uint32 [[cexport]]:
    return counter
endfunc
```

Native code can refer to `dq_counter` and `ReadCounter` using the corresponding C declarations. Match the types and calling convention described in C Interoperability.

`[[used]]` retains a function or global value even when ordinary DQ code does not refer to it. Use it for names reached through assembly, a linker script, or a hardware table. It preserves the definition through compilation and linking, including link-time optimization.

`[[section('name')]]` puts a function or global in a named native section. A linker script or other build setup may be needed to place it at a particular hardware address.

2.20.5 Layout Attributes

`[[packed]]` removes ordinary layout padding from a structure or compound type. `[[align(n)]]` requests alignment on supported global storage or fields:

```
struct SHeader [[packed]]:
    kind : uint8
    length : uint32
endstruct
```

Use packed layout only when the file format, hardware, or external declaration requires it. Packed fields may be unaligned. Check `SizeOf` and `OffsetOf` instead of assuming a particular layout, and choose alignment values supported by the target. See Structures and Unions.

2.20.6 Volatile and Register Access

A hardware register is a memory location through which a program communicates with a device. Reading it may observe changing device state; writing it may start an operation. `volatile` tells the compiler to preserve those accesses even when ordinary memory optimizations could remove them.

Hardware registers often live at fixed memory addresses. Register attributes let a declaration say whether each register can be read, written, or both:

```
struct SDeviceRegisters:
    STATUS  : [[regro]] uint32
    CONTROL : [[regrw]] uint32
    COMMAND : [[regwo]] uint32
endstruct
```

This structure describes the register layout; declaring the type alone does not connect it to a device. Code for a particular board must obtain the correct base address before accessing its fields through a pointer.

The register attributes expand to the fundamental access attributes:

Register attribute	Equivalent attributes	Allowed operations
`[[regrw]]`	`[[volatile]]`	read and write
`[[regro]]`	`[[volatile, nowrite]]`	read only
`[[regwo]]`	`[[volatile, noread]]`	write only

Reading `[[noread]]` storage or writing `[[nowrite]]` storage is a compile error. A modify-assignment such as `|=` reads and writes its target and therefore needs both operations to be allowed. `noread` and `nowrite` cannot be combined on the same declaration. These restrictions apply to direct global, field, and fixed-array-element access and are retained through module interfaces.

2.20.6.1 Access Limitations

Volatile accesses are observable compiler operations, but they are not atomic and do not imply a CPU or compiler memory barrier. Taking an address or binding a reference currently discards `noread` and `nowrite`; this is an explicit low-level escape. Whole-structure initialization and copying also do not yet apply the access restrictions of individual fields.

2.20.7 Weak Attribute

`[[weak]]` provides a fallback function that native code can replace with a strong definition of the same linker name. Combine it with `[[cexport]]` when C or assembly code should use the function's plain DQ name:

```
func DefaultHandler() [[weak, cexport]]:
    // used unless another linked definition replaces DefaultHandler
endfunc
```

`weak` does not change the name by itself. It takes no arguments and applies only to module-level function definitions, not external declarations or methods.

2.20.8 Validity and Conflicts

Unknown, unsupported, or misplaced attributes produce warnings or errors. Resolve those diagnostics rather than relying on an attribute the compiler could not apply. An attribute cannot bypass a language rule: for example, an `external` function cannot also have a DQ body.

Some combinations have additional restrictions. `weak` requires a module-level definition; assembly functions have the combinations described in Assembly Functions. Access restrictions apply to reads and writes individually, so an operation such as `+=` requires both.

For object method attributes, see Inheritance and Virtual Dispatch.

2.21. Compiler Directives

A **compiler directive** controls how source code is compiled. Use directives to enable optional logging, choose code for a target platform, include shared source, or link an external library. Directives are processed during compilation; they do not run each time the program runs.

For example, include a diagnostic message only when `LOGGING` is defined:

```
use print

#define LOGGING
#define BUFFER_SIZE = 256

func *Main():
    #ifdef LOGGING
        PrintLn("buffer size = {}", [BUFFER_SIZE])
    #endif
endfunc
```

This prints `buffer size = 256`. Remove `#define LOGGING` and compile again to leave out the print statement entirely. By comparison, an ordinary `if` chooses what to execute while the program is running.

2.21.1 Definitions and Namespaces

`#define NAME` defines a name with the value `true`. Add `= expression` to supply a constant value, or add `: Type = expression` to specify its type:

```
#define LOGGING
#define BUFFER_SIZE = 256
#define DOUBLE_BUFFER_SIZE = BUFFER_SIZE * 2
#define BOARD_NAME : rostr = "PowerLogger-Lite"
#define MCU_RK3506 : bool = false
#define EXTERNAL_XTAL_HZ : uint = 24000000
```

Typed definitions follow the same type, conversion, and constant-initializer rules as `const`; the resulting name belongs to `@def`. Types and initializer names resolve in the surrounding scope. Without a type annotation, the type is taken from the constant expression. The bare flag form is equivalent to `#define NAME : bool = true`.

You can use earlier definitions in later ones. Definitions can also come from compiler options such as `-DLOGGING` or `-DBUFFER_SIZE=512`, project files, and target settings.

Definitions belong to the `@def` namespace and are normally available by their short names. Use `@def.BUFFER_SIZE` if a local declaration has the same name. A `use` makes the imported module's public definitions available in `@def`. Imported definitions are visible within the importing module, but reach its callers only when that import uses `reexport`. For example, if A uses B and MAIN uses A, MAIN sees B's definitions only if A reexports B. Definitions declared after `implementation` stay private to their module.

This works with cached modules. Namespace-only and selective imports also make definitions available within the importing module. Repeated imports keep the definition already in scope. Command-line, project, and target definitions are build inputs available to every module; they are not exported by modules.

Use `const` for ordinary program data such as clock frequencies, limits, strings, and lookup tables. Use `#define` for configuration and feature selection, such as MCU families, board capabilities, and optional functionality:

```
#define MCUF_RK
const MAX_CLOCK_SPEED : uint = 200000000
```

A typed `#define` is useful when the value itself is a configuration setting. Both constants and definitions can be used in expressions and conditional compilation. Definitions provide constant values, not C-style textual macro substitution.

2.21.2 Conditional Compilation

`#if` tests a constant boolean expression. `#ifdef` tests whether a name exists. Use `#else` for an alternative and `#endif` to close the group:

```
use print

func *Main():
    #ifdef WINDOWS
        PrintLn("Running on Windows")
    #elifdef LINUX
        PrintLn("Running on Linux")
    #else
        PrintLn("Running on another platform")
    #endif
endfunc
```

Only the selected branch becomes part of the program. Other branches are skipped before normal DQ parsing, although their nested conditional directives must still balance. This lets a source file contain declarations or assembly instructions that are valid only for the selected platform.

Directive	Effect
`#if expression`	Start a branch selected by a constant Boolean expression.
`#ifdef name` / `#ifndef name`	Start a branch if a visible name exists / does not exist.
`#elif expression`	Test another condition if no earlier branch was selected.
`#elifdef name` / `#elifndef name`	Test another name-existence condition.
`#else`	Select the remaining alternative.
`#endif`	Close the conditional group.

For a value comparison, use `#if` rather than `#ifdef`:

```
#define API_LEVEL = 4

#if API_LEVEL >= 3
    const HAS_NEW_API : bool = true
#else
    const HAS_NEW_API : bool = false
#endif
```

The condition must be a compile-time `bool` and can use only names already visible at that point. A variable whose value is known only at runtime cannot supply the condition.

2.21.3 Compile-Time Messages

`#error "message"` emits an error and causes compilation to fail. `#warning "message"` emits a warning and allows compilation to continue. Their diagnostic identifiers are `CDirError` and `CDirWarning`, respectively.

Messages use the same string syntax, escapes, and constant interpolations as ordinary string literals, including format specifications:

```
#define API_LEVEL = 2

#if API_LEVEL < 3
    #error "API level \{API_LEVEL} is too old; level 3 is required"
#endif

#warning "Building with API level \{API_LEVEL:04X}"
```

Interpolation names must already be visible at the directive's location, and their values must be constant. Both directives are ignored in inactive conditional branches. Inline forms such as `#{warning "message"}` are also supported.

2.21.4 Target and Feature Definitions

The compiler supplies names for the selected target and enabled features:

Definition	Meaning
`LINUX`, `WINDOWS`, `WASI`, `BARE`	Effective platform/runtime selection.
`X86_FAMILY`, `X86`, `X64`	x86 family and selected 32/64-bit architecture.
`ARM`, `ARM_M`, `ARM_A`, `ARMHF`	Applicable Arm profile/ABI markers.
`ARM64`, `AARCH64`	AArch64 target markers.
`RISCV`, `RV32`, `RV64`	RISC-V family and register width.
`WASM`	WebAssembly target.
`TARGET_32BIT`, `TARGET_64BIT`	Effective pointer width.
`PTRSIZE`	Pointer size in bytes.
`EXCEPTIONS`	DQ exception support is enabled.
`DYNSTRINGS`	Heap-backed dynamic strings are enabled.
`HARDFLOAT`	Hardware floating-point instructions are supported (32-bit or 64-bit).
`HARDFLOAT32`	Hardware 32-bit (`float32`) floating-point support is available.
`HARDFLOAT64`	Hardware 64-bit (`float64`) floating-point support is available.

Check the feature you need rather than assuming every target has it. Disabled features are normally absent, so use `#ifdef EXCEPTIONS`, for example, to test for exception support.

2.21.5 Include

`#include` inserts another file's source at that point:

```
#include "./shared.dqi"
```

Single quotes are also accepted. Use an include for a source fragment; use `use` when you want a module with its own public interface and namespace.

Include paths support the same compile-time string interpolation as ordinary string literals. Expressions can use constants already visible at the directive:

```
const PLATFORM_INC : rostr = "./platform/config.dqi"
#include "\{PLATFORM_INC}"
```

Interpolation is evaluated before resolving the path. Runtime values cannot select an include, and includes in inactive conditional branches are not evaluated.

Local include paths must start with `./` or `../`; bare filenames are rejected. A path such as `platform/config.dqi` selects the highest-priority `platform` package directory first, then looks for `config.dqi` inside it. A missing file is an error; lookup does not fall back to another version of the package or search the current directory. Include paths retain their filename extensions. `./` and `../` are relative to the source file; `^/` is relative to the configured package containing the current source file, just like `use ^/`. Inside an included fragment from another package, this is the fragment's package root; returning to the including file restores its context. A `^/` path requires the current source file to belong to a configured package.

Package search paths also configure include discovery: `--pkg_path` and the project-file `pkg_path` setting, as well as direct `--pkg` / `pkg` mappings, apply to both `use` and package-qualified `#include` paths. No separate include-directory list is needed. A platform package can ship configuration fragments, register modules, and drivers together; the project only needs to make that package discoverable. For example, with `pkg_path = './packages'`, `#include "platform/config.dqi"` selects the file `packages/platform/`

`config.dqi` relative to the project directory. Within that fragment, `./` and `../` resolve from the fragment's directory, while `^/` resolves from the selected `platform` package root.

2.21.5.1 Include-Once Fragments

Put `#include_once` at the start of an included fragment when it should be processed only once. Later inclusions skip the rest of that file. This is separate from `use`, which already imports a module as one shared instance. For example, save this fragment as `shared.dqi` beside your main file:

```
#include_once
const MAX_ITEMS : int = 16
```

The main file can include it more than once, directly or through other fragments, without declaring `MAX_ITEMS` twice. Included declarations appear in the scope where the include occurs; the fragment has no separate module namespace.

2.21.5.2 Separate Header Files

A module can keep its public declarations in a `.dqh` file with the same base name. For `file.dq`, this directive includes `file.dqh`:

```
#include header
```

Use it before `implementation`. It cannot be used from inside a `.dqh` file.

2.21.6 Link Libraries

`#linklib` asks the linker to use an external library. For example, the following declaration gets zlib's version string:

```
#linklib('z')

func zlibVersion() -> ^char [[external]] ..
```

The library must be installed and available to the target's linker. See C Interoperability for the full example and setup links.

2.21.7 Existence Tests and Defined

Existence tests find visible names, including local variables, parameters, object members, types, module declarations, merged imports, and definitions. `Defined(NAME)` provides the same test as a constant boolean expression:

```
#define LOGGING = false

#if Defined(LOGGING) and @def.LOGGING
    // included only if LOGGING exists and its value is true
#endif
```

A name can exist even when its value is `false` or zero. `#ifdef LOGGING` therefore succeeds for the definition above; it does not read that value.

Qualified names restrict lookup: `@def.FEATURE` checks definitions, `@.MODULE_VALUE` checks the current module, and `@module.PUBLIC_VALUE` checks that namespace. Names from inner or completed sibling scopes are not visible. Preprocessor definitions remain available to existence tests in methods even without a method-local `use`.

You may test whether an uninitialized local exists without reading it. This does not permit you to read the variable before assigning a value.

2.21.8 Optional Constants

`FirstInt`, `FirstUint`, `FirstFloat`, and `FirstBool` select the first available constant from optional names, with a required fallback at the end. This is useful for settings that a board or build configuration may supply:

```
const BUFFER_SIZE : int = FirstInt(BOARD_BUFFER_SIZE, DEFAULT_BUFFER_SIZE, 256)
const CLOCK_HZ : uint = FirstUint(BOARD_CLOCK_HZ, DEFAULT_CLOCK_HZ, 8000000)
const SCALE : float = FirstFloat(BOARD_SCALE, 1)
const LOGGING : bool = FirstBool(BOARD_LOGGING, false)
```

Missing names are skipped from left to right. The first name that exists must be a constant convertible to the required type. A variable or an incompatible constant is an error, rather than a reason to try the next name. The fallback must be valid even when it is not selected.

The results are `int`, `uint`, `float`, and `bool`, respectively. These expressions always run during compilation and can be used in ordinary expressions or conditional directives. Optional candidates must be bare names, such as `BOARD_SCALE`, rather than qualified names or arbitrary expressions.

2.21.9 Directive Forms

An ordinary directive starts with `#` and ends at the physical line ending. Use `#{...}` to embed a directive inside a source line or to span multiple lines. The closing `}` ends the directive:

```
#define FAST
var value : int = #{ifdef FAST} 1 #{else} 2 #{endif}

#{if Defined(MCU_STM32G474) or Defined(MCU_STM32G473)
     or Defined(MCU_STM32G474RE)
}
    const HAS_G4 : bool = true
#endif
```

Directives accept `//` and `/* ... */` comments. A line comment does not extend a directive onto the next line. Only supported directive names have meaning; there is no general `#pragma` or implied syntax from other languages.

2.21.10 Include Dependencies

If an implementation include affects the compiled module, declare it before `implementation` with `#srcdep`:

```
#srcdep "./somefunc_impl.dqi"

func Calculate(value : int) -> int ..

implementation

#include "./somefunc_impl.dqi"
```

Here `somefunc_impl.dqi` contains the body of `Calculate`. `#include` inserts that body where it belongs, while the earlier `#srcdep` tells the build system that a change to the fragment can require recompiling the module.

`#srcdep` uses the same explicit local and package path rules as `#include`. It records the dependency without inserting its contents there. It uses the same path lookup as `#include` and records only the named file; it does not parse that file or discover its dependencies.

Automatic include tracking covers all files included before `implementation`, including nested includes from `.dqi` fragments and the same-basename `.dqh`. Changing any of these files invalidates the module's cached artifacts. Includes in inactive conditional branches are not processed or recorded. For files included after `implementation`, declare each dependency with `#srcdep` before `implementation`.

For build-option precedence and library search paths, see Compiler Usage and Options and Project Files.

2.22. C Interoperability

DQ can call functions from C libraries and let C code call DQ functions. This is useful for reusing existing libraries or working with an operating system's native APIs. There are two parts to calling a C function: **declare** its interface so DQ can compile the call, and **link** its implementation so the finished program can run it. Declare the function's name, parameter types, and result in DQ, then mark it `[[external]]`.

For example, call the C library's `puts` function to print a line:

```
func c_puts(text : ^char) -> int32 [[external('puts')]] ..

func *Main():
    c_puts("hello from C")
endfunc
```

This prints `hello from C`. `external('puts')` connects the DQ name `c_puts` to the native function named `puts`. Normal hosted programs already link the C runtime; other libraries may need an explicit link request.

2.22.1 External Functions and Symbols

If the DQ and native names are the same, use plain `[[external]]`:

```
func puts(text : ^char) -> int32 [[external]] ..
```

The trailing `..` ends the declaration without a DQ body. The implementation comes from the external library.

Match the library's actual declaration, usually found in its C header. The compiler checks your DQ code against the declaration you wrote; it cannot check that declaration against the library. A wrong signature may compile successfully and still fail at runtime.

The **ABI**, or application binary interface, includes details such as type sizes, memory layout, and how function arguments are passed. These must agree between DQ and the native code.

2.22.2 Match C Types Explicitly

Use fixed-width DQ types where the C declaration has a known width:

C-side concept	Typical DQ declaration
`int`, `unsigned int` on usual hosted ABIs	`int32`, `uint32`
Fixed-width `int8_t` … `uint64_t`	Matching DQ fixed-width integer.
`size_t`	`uint` on supported pointer-sized hosted ABIs.
`float`, `double`	`float32`, `float64`
`void *`, opaque handle	`pointer`
Pointer to a known type	`^T`
Zero-terminated byte string	`^char` or a compatible `rostr` argument.
C record/union	An explicitly matching `struct` / `union` layout.

DQ `int` is pointer-sized, so use `int32` for the usual C `int`. C `long` also varies by platform: it has different widths on Windows x64 and Linux x64. Check the target's header rather than assuming similar type names match.

DQ objects, dynamic arrays, `str`, and unsized `embstr` are not automatically represented like a C library's objects, arrays, or strings. Pass the raw data in the form the library expects.

2.22.3 Strings, Buffers, and Borrowing

C string functions usually expect a pointer to bytes ending in a zero byte. A text literal or `rostr` can be passed directly to a `^char` parameter:

```
use print

func strlen(text : ^char) -> uint [[external]] ..

func *Main():
    var text : rostr = "hello"
    PrintLn("{} bytes", [strlen(text)])
endfunc
```

This prints `5 bytes`. DQ passes the underlying text address without copying it. Although the C parameter is written `^char`, do not let C write through it when the source is read-only.

For a dynamic `str` or fixed-storage `embstr(N)`, use `.pchar` to obtain a borrowed pointer to terminated text. A `strslice` may not end in a zero byte; copy it into a `str` or fixed-storage buffer before passing it to a C string function:

```
var message : str = "hello world"
var first_word : str = message[:5]
c_puts(first_word.pchar) // prints hello, using c_puts from the opening example
```

Keep the text or buffer alive while C uses its address. If the library stores the pointer for later, the source must remain valid for that entire period.

A C function that writes into a buffer must know its usable size. Do not let it write beyond the capacity or bypass DQ's string length and sharing rules. See Buffer and String Details before passing writable string storage.

2.22.4 Variadic Functions

A C function with `...` accepts extra arguments. For example, `printf` uses its format string to interpret those arguments:

```
func printf(format : ^char, ...) -> int32 [[external]] ..

func *Main():
    var value : int32 = 42
    printf("value = %d\n", value)
endfunc
```

This prints `value = 42`. Here `%d` expects a C `int`, so the argument is `int32`. DQ cannot verify the types expected by a C format string. Extra arguments follow the target's C argument-promotion rules. DQ's own formatting instead uses a checked list of `anyvalue` values.

2.22.5 Callbacks

A **callback** is a function you give to a library for it to call. Use a plain function reference with the exact signature the C API expects. A bound `of object` method reference also stores its receiver, so it is not a plain C function pointer. Keep any callback context alive for as long as the library may use it.

For example, the C library's `qsort` function sorts a buffer by repeatedly calling a comparison function supplied by the caller:

```
use print

type FCompare = func(left : pointer, right : pointer) -> int32

func qsort(base : pointer, count : uint, size : uint,
    compare : FCompare) [[external]] ..

func CompareInts(left : pointer, right : pointer) -> int32:
    var a : int32 = ^int32(left)^
    var b : int32 = ^int32(right)^
    if a < b:
        return -1
    elif a > b:
        return 1
    endif
    return 0
endfunc

func *Main():
```

```
    var values : [4]int32 = [30, 10, 40, 20]
    qsort(%values[0], values.length, SizeOf(int32), CompareInts)
    for value : int32 in values:
        PrintLn("{}", [value])
    endfor
endfunc
```

This prints `10`, `20`, `30`, and `40`, each on its own line. `qsort` receives the address of the first element, the number of elements, and the size of each one in bytes. It passes two element addresses to `CompareInts`, which reads them as `int32` values and returns a negative, zero, or positive C `int`. Explicit comparisons avoid the overflow that subtracting the values could cause.

`qsort` rearranges the caller's array during the call and does not retain its address afterward. The local array therefore lives long enough. This example uses plain fixed-width numbers; do not use raw C byte-copying operations to rearrange DQ values that require managed copying or cleanup.

2.22.6 Linking and Native Ownership

Use `#linklib('name')` when an external library needs to be linked explicitly. For example, this program prints the installed zlib version:

```
use print
#linklib('z')

func zlibVersion() -> ^char [[external]] ..

func *Main():
    var version : rostr = zlibVersion()
    PrintLn("zlib {}", [version])
endfunc
```

This requires zlib to be installed in the target linker's search path. Compiler options and project files can also provide native libraries and object files. Existing C declarations are available through modules such as `use libc/stdio`.

Release resources with the operation their API specifies. For example, close a native file using the library's close function. A returned pointer may be borrowed, or may need a library-specific release function; do not assume that `delete` or an `autofree` pointer is appropriate.

Linux setup and Windows setup explain native library installation and search paths.

2.22.7 Exporting DQ Functions

Use `[[cexport]]` to expose a function under its DQ name, or `[[export('name')]]` to choose its native name:

```
func Add(left : int32, right : int32) -> int32 [[cexport]]:
    return left + right
endfunc
```

The corresponding C declaration includes the header for fixed-width integers:

```
#include <stdint.h>

int32_t Add(int32_t left, int32_t right);
```

The C and DQ files must be compiled for the same target and linked together. The export attribute supplies the symbol name; it does not itself build a shared library or arrange runtime initialization for a C-owned entry point. See Attributes for exported globals and symbol retention.

Avoid letting a DQ exception cross a native call boundary unless that interface explicitly supports it. Handle failures within the DQ entry point when needed.

2.22.8 External Global Variables

A global variable may also come from native code:

```
var native_counter : int32 [[external('native_counter')]]
```

The linked native code must supply that symbol. Some C headers expose apparent globals through macros or functions instead. For example, do not assume `errno` is an exported integer variable; use the platform's real interface.

2.22.9 Structures, Unions, and Layout

When passing a C record or union, match its field types, order, padding, alignment, and calling convention. Check `SizeOf` and `OffsetOf` on the target. Use `[[packed]]` only when the corresponding C record is packed the same way; packing a normally aligned record changes its layout.

A union gives several fields the same storage without remembering which one was written. Ensure the selected field contains a valid value before reading it. See Structures and Unions and Attributes.

2.22.10 Buffer and String Details

`embstr(N)` has `N` storage bytes including its zero terminator, leaving room for at most `N - 1` content bytes. A C writer must respect both that capacity and any runtime descriptor requirements. Raw writes into a dynamic `str` must not bypass its copy-on-write and length handling.

C termination-based APIs see only the text before the first zero byte. To pass binary data or text containing embedded zeros, use an API that takes a pointer and a length. DQ text operations preserve embedded zeros when the length is known.

`rostr` also converts to `^char` in C variadic arguments, such as a `%s` argument to `printf`. Neither this conversion nor `.pchar` extends the source lifetime. See Conversions and Borrowed Lifetimes.

2.22.11 WebAssembly Imports

For WebAssembly functions, `[[external('name', 'module')]]` can select both the import name and its module. The second argument applies only to function imports on that target, not to external global variables.

2.23. Assembly Functions

An assembly function lets you write machine instructions and call them like a DQ function. Use one when you need a specific processor instruction that DQ expressions or library functions do not provide. You will need to know the assembly language for your target processor.

Here is a complete **x86-64** example that adds two numbers:

```
use print

func Add(left : int, right : int) -> int [[inline, asm]]:
    lea $result, [$left + $right]
endfunc

func *Main():
    PrintLn("{}", [Add(19, 23)])
endfunc
```

This prints `42`. `$left` and `$right` stand for the function arguments, and `$result` stands for its return value. The compiler chooses the actual registers. For ordinary addition, use a DQ expression; this small example shows how an assembly operation connects to a normal function call.

There are two forms:

Attributes	What you write
`[[inline, asm]]`	Instructions inserted at each direct call; the compiler connects arguments and results.
`[[asm]]`	A complete native function, including argument handling and return instructions.

Both use a colon body closed by `endfunc`. They do not accept a bodyless declaration or a brace body. Instructions and register names depend on the target; the x86-64 examples do not work unchanged on ARM or RISC-V.

2.23.1 Inline Assembly Functions

`[[inline, asm]]` is useful for a small operation that should become part of the caller's code. Declare its input and result types just as for an ordinary function, then use named operands in the instructions.

On x86-64, use Intel syntax. ARM, AArch64, and RISC-V use their native syntax. DQ does not translate instructions or reorder their operands.

2.23.1.1 Named Operands

Use `$result` for the return value and `$parameter` for a function parameter. The compiler replaces these names with the target assembly operands allocated for the call.

For example, ARM targets with the `rbit` instruction can reverse the bits of a 32-bit value:

```
func ReverseBits(value : uint32) -> uint32 [[inline, asm]]:
    rbit    $result, $value
endfunc
```

`$result` is valid only when the function has a return type. An unknown name is a compile error. A literal dollar sign in assembly text must be written as `$$` because the underlying assembly template syntax reserves `$`.

By default, parameters are register inputs and the result is a register output. Use the hints below for constant operands, memory access, or additional effects. The input and output registers may overlap. Write instructions that remain correct if `$result` uses the same register as an input, and do not modify an input register unless that change is represented by an output or clobber.

2.23.1.2 Operand Hints

The compiler must know what your instructions read or change. Put one hint list immediately before `endfunc` to describe those effects. For example, this x86-64 function multiplies by a compile-time constant:

```
func MulImmediate(value : int32, factor : int32) -> int32 [[inline, asm]]:
    imul    $result, $value, $factor
    [[immediate(factor), clobber(flags)]]
endfunc
```

The hint list tells the compiler about the instructions; it is not assembly text. It supports these clauses:

Hint	Meaning
`immediate(parameter, ...)`	Require compile-time integer-like arguments and use them as immediate operands
`memread(parameter, ...)`	Describe memory read through typed pointer or readable reference arguments
`memwrite(parameter, ...)`	Describe memory written through typed pointer or writable reference arguments
`memreadwrite(parameter, ...)`	Describe memory both read and written through typed pointer or read/write reference arguments
`clobber(resource, ...)`	Declare registers or other target resources modified by the assembly

A clause can name multiple operands. Clause and argument order do not affect the meaning. Each parameter may occur in at most one hint; parameters omitted from the list remain register inputs.

An `immediate` argument must be a compile-time constant at every call site. Its type must be an integer, Boolean, character, or enumeration type.

Memory operands must be typed pointers or references so that the compiler knows the accessed element type. The reference mode must permit the described access: `refin` can be read but not written, `refout` can be written but not read, and `ref` supports all three memory hints. The programmer remains responsible for passing a valid address.

A **clobber** names a register or other resource changed by the instructions. `clobber(memory)` is supported on every inline-assembly target. It acts as a compiler barrier, preventing memory operations from being moved across it:

```
func CompilerBarrier() [[inline, asm]]:
    [[clobber(memory)]]
endfunc
```

This does not emit a processor memory-barrier instruction.

`clobber(flags)` declares the condition-code state on x86, ARM, and AArch64 targets. It is not available on RISC-V. A physical register can also be named as a clobber, using its lowercase target name or a recognized conventional alias:

```
func AddOneThroughRax(value : int) -> int [[inline, asm]]:
    mov     rax, $value
    lea     $result, [rax + 1]
    [[clobber(rax)]]
endfunc
```

Declare every memory access and every modified resource that is not already an output. The optimizer can only account for effects described by the signature, memory hints, and clobbers.

2.23.2 Reading and Writing Memory

Use memory hints for instructions that access a pointed-to value. With these hints, `$address` means the memory at that address, rather than a register containing the pointer. These examples use x86-64 Intel syntax:

```
func Load32(address : ^int32) -> int32 [[inline, asm]]:
    mov     $result, $address
    [[memread(address)]]
endfunc

func Store32(address : ^int32, value : int32) [[inline, asm]]:
    mov     $address, $value
```

```
    [[memwrite(address)]]
endfunc

func AddTo32(address : ^int32, value : int32) [[inline, asm]]:
    add     $address, $value
    [[memreadwrite(address), clobber(flags)]]
endfunc
```

To try these three functions, place `use print` before them and add:

```
func *Main():
    var value : int32 = 10
    Store32(%value, 20)
    AddTo32(%value, 3)
    PrintLn("{}", [Load32(%value)])
endfunc
```

This prints `23`. `%value` supplies the address of the local variable. `memwrite` describes the store, `memreadwrite` describes the addition, and `memread` describes the load. These hints tell the optimizer how the assembly interacts with the surrounding DQ code; the instructions perform the actual reads and writes.

An instruction may still require an explicit size qualifier when its other operands do not determine the access width:

```
func Inc32(address : ^int32) [[inline, asm]]:
    inc     dword ptr $address
    [[memreadwrite(address), clobber(flags)]]
endfunc
```

2.23.3 Complete Assembly Functions

Use `[[asm]]` when you need to control the whole native function. Unlike the inline form, its body must include the instruction that returns to the caller.

```
use print

func Return42() -> int [[asm]]:
    mov     eax, 42
    ret
endfunc

func *Main():
    PrintLn("{}", [Return42()])
endfunc
```

This prints `42` on x86-64 and uses Intel syntax. DQ uses the unified `instruction destination, source` operand order and does not support AT&T syntax.

The DQ signature determines the calling convention: the rules for passing arguments and returning values. The assembly implementation must read arguments from the correct registers or stack locations, place a return value in the correct location, preserve all required registers, and return to the caller. Parameter names and `result` are not available as operands in a complete assembly function.

The compiler emits an `[[asm]]` function as naked and non-inline. Consequently, it does not generate a prologue or epilogue (the instructions that normally set up and restore a function's stack and registers), or a DQ `return` operation. `[[asm]]` cannot be combined with `[[external]]`, `[[abstract]]`, or `[[always_inline]]`.

2.23.4 Inline Assembly Restrictions

An inline assembly function does not produce a separately callable function. Every direct call emits the assembly template at the call site. It therefore cannot be converted to a function reference or have its address taken.

Inline assembly functions must be ordinary module-level functions. Methods and special functions such as `*Main` and `*ModuleInit` are not supported. They also cannot use `[[external]]`, `[[noinline]]`, `[[weak]]`, `[[section]]`, or native export attributes. `[[overload]]` is allowed, and inline assembly definitions in a module's public interface remain available to importing modules.

2.23.4.1 Operand Types and Registers

Without an operand hint, each parameter is a register input. Integer, Boolean, character, enumeration, pointer, and floating-point scalar operands are supported. The result is always a register output and must also have a supported scalar type. Integer and pointer values use the target's general-purpose register class; floating-point values use its native floating-point register class. Availability of a particular floating-point register class may depend on the selected CPU features.

2.23.5 Source Processing

Assembly bodies participate in normal DQ source processing:

- `//` and `/* ... */` comments are removed;
- compiler directives are processed, and inactive conditional branches are omitted;
- `endfunc` closes the body only when it is active and outside a comment;
- comments and skipped branches preserve the separation between assembly tokens and instructions.

This permits target-specific implementations selected with conditional compilation. This x86-64 example uses a build-supplied `TARGET_HAS_PAUSE` definition to choose the instruction:

```
func Pause() [[inline, asm]]:
    #ifdef TARGET_HAS_PAUSE
        pause
    #else
        nop
    #endif
endfunc
```

Inside an assembly body, `#` starts a DQ directive only when it is the first non-whitespace, non-comment character on a physical source line. After assembly text has started on the line, `#` remains target assembly syntax. For example, the immediate in `mov r0, #1` is preserved. Keep an instruction and any line-leading immediate operand on the same physical line; a continuation line starting with `#` would be parsed as a directive.

Because DQ comments are removed before assembly is parsed, do not use `//` or `/* ... */` when those characters are intended as target assembler syntax. A line-leading `#` likewise cannot be used as a target assembler comment. In a complete `[[asm]]` body, `[[...]]` has no special meaning. In an `[[inline, asm]]` body, a line beginning with `[[` is parsed as the single trailing operand-hint list, after which only comments, whitespace, and `endfunc` may follow.

2.23.6 Supported Inline-Assembly Targets

Typed inline assembly is supported for these target identifiers:

- x86-64: `x86_64`;
- ARM: `arm_m0`, `arm_m3`, `arm_m4`, `arm_m4f`, `arm_m33`, `arm_m7f`, `arm_m7fd`, `arm`, and ARM-A identifiers beginning with `arm_a`;
- AArch64: `aarch64`, `arm64`;
- RISC-V 32-bit: `riscv32`, `rv32i`, `rv32imac`;
- RISC-V 64-bit: `riscv64`, `rv64g`.

Complete `[[asm]]` functions are not restricted to this list. In either form, instructions must match the selected target and CPU features.

3. DQ Compiler and Tools

Install the tools for your host platform, then use `dq-run` for small programs or `dq-comp` and a project file for a repeatable application build.

- Compiler and Tool Overview
- Runtime and Program Execution
- Linux Development Environment
- Windows Development Environment
- Compiler Usage and Options
- Project Files
- Build Process and Module Artifacts
- Targets and Cross-Compilation
- VSCode Setup and Debugging

Compiler contributors can continue with Building the Compiler from Source.

3.1. Compiler and Tool Overview

The DQ tools turn `.dq` source modules into native programs. The compiler uses LLVM to generate machine code and a Clang linker driver to combine that code with the DQ runtime and native libraries.

Tool	Purpose
`dq-comp`	Compile a source module or `.dqproj` project and link an executable.
`dq-run`	Compile and run a single source file.
`dq-atrun`	Compile and execute the repository's autotests.

3.1.1 Install and Check the Tools

Use a DQ release package for your host platform. Keep its `bin`, `stdpkg`, and bundled toolchain directories together, since the tools discover supporting files relative to the compiler.

Follow Linux Development Environment or Windows Development Environment for installation. To work on the compiler itself, see Building the Compiler from Source.

```
dq-comp --version
dq-comp --help
dq-run --help
```

3.1.2 Compile a Program

Save this as `hello.dq`:

```
use print

func *Main():
    PrintLn("Hello, DQ!")
endfunc
```

Run it directly:

```
dq-run hello.dq
```

Or keep the compiled executable:

```
dq-comp -g -O0 hello.dq
./hello
```

On Windows, the output is `hello.exe`; run it with `.\hello.exe` in PowerShell. `*Main` is the DQ program entry point. The selected runtime supplies the native startup code on hosted targets.

3.1.3 Choose a Workflow

Use `dq-run` for small examples. Compiler options go before the source filename; program arguments go after it, optionally separated by `--`:

```
dq-run -O2 hello.dq -- first second
```

When no compiler options are supplied, `dq-run` selects `-g -O0`. These flags provide source information for debugging and runtime backtraces.

Use `dq-comp` for explicit output names, reusable builds, and cross targets. A project file keeps a longer set of options together. Imported DQ modules and runtime modules are compiled as needed; you do not need to list every imported source file on the command line.

The next chapters describe program execution, compiler options, and the generated build artifacts.

3.2. Runtime and Program Execution

The DQ runtime is written in DQ and lives in `stdpkg/rtl`. It supplies the services needed by generated code: managed storage, runtime checks, module initialization, and platform startup. The compiler builds the required runtime modules alongside your program.

3.2.1 Hosted Startup

On Linux, Windows, and WASI, the selected runtime supplies the native `main` entry point. Startup records the command-line arguments, initializes imported modules in dependency order, then calls the DQ `func *Main()` entry point. Each canonical module is initialized once.

Returning from `*Main` ends a normal hosted run. The implicit `exit_code` variable starts at zero; assign it to select the process result while allowing local scope cleanup to finish:

```
use print

func *Main():
    if ArgCount() < 2:
        PrintLn("Please supply a name")
        exit_code = 1
        return
    endif
    var name : rostr = ArgStr(1)
    PrintLn("Hello, {}!", [name])
endfunc
```

Run the saved program with `dq-run hello.dq -- Alice`. `ArgCount()` includes the program name at index zero. `ArgStr(i)` returns a borrowed zero-terminated pointer, or `nil` when the index is outside the argument list.

`Exit(status)` terminates a hosted process immediately through the C runtime. Use a normal return when local destructors and `finally` actions need to run. DQ does not currently perform automatic global/module finalization at shutdown; release resources explicitly when their shutdown actions matter.

3.2.2 Implicit Runtime Interface

Ordinary modules automatically receive the public names of `rtl/sys`. These include memory helpers, string helpers, and, on hosted targets, argument access. The exception interface is included when exceptions are enabled. The `sys` chapter lists the user-facing services.

`--no-use-sys` suppresses that implicit import. It is intended for bootstrap and low-level runtime modules that must control their own dependencies.

3.2.3 Runtime Features

Feature	Hosted default	Bare default	Selection
Exception handling	Enabled, except on WASI	Disabled	`--exceptions` / `--no-exceptions`, or `exceptions` in a project.
Dynamic strings	Enabled	Disabled	`--dynstrings` / `--no-dynstrings`, or `dynstrings` in a project.

The compiler defines `EXCEPTIONS` and `DYNSTRINGS` only when the corresponding feature is enabled. Use conditional compilation for code that needs both configurations.

With exceptions disabled, `try`, `except`, `finally`, and `raise` are rejected. Runtime checks remain active and report failures through `RuntimeError` and its runtime error hook. Hosted defaults print the error and terminate the process.

With dynamic strings disabled, heap-owning `str` operations are rejected. Literals, `rostr`, `strslice`, and fixed-storage `embstr` remain available. See Project Files for feature settings.

3.2.4 Exceptions and Diagnostics

On exception-enabled hosted targets, an uncaught exception reaches the runtime's top-level reporter, which prints the message and backtrace and exits with failure. Compile with `-g -O0` to retain useful source locations:

```
dq-comp -g -O0 app.dq
dq-run app.dq
```

The second command chooses those flags when no compiler options are supplied. See Exceptions and Error Handling for handlers and cleanup. Signal/access-violation conversion has platform and optimization limitations documented in Exceptions and Backtraces.

3.2.5 Bare-Metal Startup

Bare targets produce objects by default and do not supply a complete board startup environment. A board project provides its entry point, reset/startup code, memory layout, and linker script. Do not assume hosted argument access, process termination, or an operating-system exception reporter.

On bare targets, the default `Exit` implementation loops indefinitely and `Errno()` returns `-1`. Runtime integration belongs to the board support code. See Targets and Cross-Compilation and bare-metal project runtimes.

3.3. Linux Development Environment

This chapter sets up a Linux computer for developing DQ applications. The package commands below use Ubuntu; other distributions may use different package names and installation commands.

3.3.1 Download and Extract DQ

Download the Linux compiler archive for your CPU architecture from the DQ releases page. For an x86_64 computer, the full package is named `dq-VERSION-x86_64-linux-full.tar.gz`, where `VERSION` is the release number.

The full package includes the DQ tools, standard packages, examples, and the Clang/LLVM toolchain used to link programs. It is the simplest starting point because it supplies the compiler and linker dependencies together.

Extract the complete archive into a directory you can keep. For the commands below, replace `VERSION` with the version in the downloaded filename:

```
tar -xzf dq-VERSION-x86_64-linux-full.tar.gz
cd dq-VERSION-x86_64-linux-full
```

Keep the package directories together. The compiler locates the standard packages and bundled tools relative to its installation.

3.3.2 Set Up the Shell Environment

From the extracted full package directory, load its environment script:

```
source ./dq-env.sh
dq-comp --version
```

This configures the current Bash shell to find the DQ tools and their bundled toolchain. `PATH` is the list of directories the shell searches for commands. You need to source the script again in a new terminal unless you install DQ or add the setup to your shell startup file.

To try an included example from the extracted package directory:

```
dq-run examples/basic/test1.dq
```

3.3.2.1 Install DQ

Both full and smaller Linux compiler archives include an installer. From the extracted archive, install for the current user with:

```
./install.sh --prefix "$HOME/.local"
export PATH="$HOME/.local/bin:$PATH"
dq-comp --version
```

Add the `export PATH=...` line to `~/.bashrc` to make it available in future Bash terminals. The installer places the tools and supporting files under `~/.local/lib/dq` and creates command symlinks in `~/.local/bin`.

For a system-wide installation, use this alternative:

```
sudo ./install.sh
dq-comp --version
```

The default installation directory is `/usr/local/lib/dq`, with command symlinks in `/usr/local/bin`. Installing a full archive also installs its bundled toolchain. Examples and the VS Code extension remain in the extracted archive, so keep it if you want to use those files.

3.3.2.2 Using a Package Without a Bundled Toolchain

If the release provides a smaller archive named `dq-VERSION-x86_64-linux.tar.gz`, it uses the system's compiler and linker dependencies. Follow that release's dependency instructions.

For a release built with LLVM 21, an Ubuntu 26.04 setup is:

```
sudo apt update
sudo apt install clang-21 lld-21 g++ libc6-dev libbacktrace-dev
export DQ_LINKER_DRIVER=clang++-21
```

The selected linker driver must match the LLVM major version used to build DQ, especially when using link-time optimization. Ubuntu 26.04 provides Clang 21. On distributions without the required version, the LLVM APT repository provides installation instructions and a script for selecting a specific version. The full DQ package avoids this additional setup.

3.3.3 Set Up Visual Studio Code

Install Visual Studio Code, then install the DQ extension shipped in the compiler archive:

1. Open the Command Palette with `Ctrl+Shift+P`.
2. Run **Extensions: Install from VSIX...**.
3. Select the `dq-syntax-*.vsix` file in the archive's `VSCode` directory.
4. Reload the VS Code window if prompted.

Open the extracted package's `examples` directory with **File > Open Folder**, then open `langdemo.dq`. It should appear with DQ syntax highlighting. Click the play button in the editor title bar or run **DQ: Run Current File** from the Command Palette.

The extension runs the current file through `dq-run` with debugging options `-g -O0`. It also starts `dq-comp` as a language server to provide compiler diagnostics and document symbols. Both commands must be available to VS Code.

If you sourced `dq-env.sh` in a terminal, start VS Code from that terminal with `code .` so it inherits the environment. After changing persistent environment settings, close and reopen VS Code.

For a portable installation that is not on `PATH`, you can set explicit tool paths in VS Code's `settings.json`:

```
{
    "dq.languageServerPath": "/home/yourname/dq/bin/dq-comp",
    "dq.runPath": "/home/yourname/dq/bin/dq-run"
}
```

Replace these example paths with your actual installation paths. See VSCode Setup and Debugging for the chapter dedicated to editor and debugger configuration.

3.3.4 Add External Libraries

Some DQ packages call external C libraries. Install the development package for each library you use so the linker can find it. For example, on Ubuntu:

```
sudo apt install libsqlite3-dev
```

For the MySQL client library:

```
sudo apt install libmysqlclient-dev
```

These are optional dependencies for applications that use the corresponding database packages. A database server is a separate installation.

3.3.5 Troubleshooting

3.3.5.1 The Shell Cannot Find `dq-comp`

Check which command the shell finds:

```
command -v dq-comp
```

Source the full package's `dq-env.sh`, or add the installation's command directory to `PATH`. If multiple versions are installed, check that the reported path is the version you intend to use.

3.3.5.2 Linking Fails

Keep the full package's `toolchain` and `stdpkg` directories with its tools. When using system tools, check `clang++-21 --version` and `ld.lld-21 --version` and select the matching driver with `DQ_LINKER_DRIVER`.

An error such as `unable to find library -lsqlite3` means the linker cannot find the requested library. Install its development package. If the problem only occurs in VS Code, check that the editor inherited the same environment as the working terminal.

3.3.6 Next Steps

Return to Getting Started to write and run your first program.

Building the DQ compiler itself requires additional C++ and LLVM development packages. That workflow belongs in Building the Compiler from Source.

3.4. Windows Development Environment

This chapter sets up Windows for developing DQ applications with the Windows UCRT compiler package. The compiler produces native `.exe` files. Its archive includes the compiler, standard packages, and the toolchain used to link programs.

The commands marked `bat` below are for **Command Prompt** (`cmd.exe`). The `sh` commands in the external-library section run inside **MSYS2 UCRT64**.

3.4.1 Download and Extract DQ

Download the **x86_64 Windows UCRT** archive from the DQ releases page. Its filename is `dq-VERSION-x86_64-windows-ucrt.zip`, where `VERSION` is the release number.

Extract the complete archive to a directory such as `C:\dq`. In this guide, `C:\dq\bin\dq-comp.exe` is the resulting compiler path. If extraction creates an additional directory containing the version number, use that directory as your DQ root instead.

Use a simple directory name without spaces, and keep all the package directories together. In particular, retain `bin`, `stdpkg`, `gcclibs`, and `toolchain`. The compiler finds its bundled linker relative to its own installation, so the toolchain directory does not need a separate `PATH` entry.

3.4.2 Make the Tools Available

For the current Command Prompt window, run:

```
set "PATH=C:\dq\bin;%PATH%"
dq-comp --version
```

The command should print the compiler version. The package also includes `dq-env.cmd`, which prepares the current Command Prompt when run from the extracted package directory.

For permanent setup:

1. Open **Edit environment variables for your account** from the Start menu.
2. Under **User variables**, select **Path** and choose **Edit**.
3. Add `C:\dq\bin`, using your actual installation path if different.
4. Confirm the dialogs and open a new terminal.

Completely close and reopen VS Code if it was running, so it inherits the new environment. `PATH` is the list of directories Windows searches for commands and runtime DLLs.

Check which compiler Windows finds:

```
where dq-comp
dq-comp --version
```

If several versions are listed, Windows normally uses the first one.

3.4.3 Compile and Run a Program

Follow Getting Started to create `hello.dq`, then open a terminal in the directory containing that file:

```
dq-run hello.dq
```

To compile and run separately:

```
dq-comp -g -O0 hello.dq
.\hello.exe
```

The Windows compiler package provides the tools needed for this basic workflow. External libraries are required only when your application uses them.

3.4.4 Set Up Visual Studio Code

Install Visual Studio Code, then install the DQ extension from the compiler archive:

1. Open the Command Palette with `Ctrl+Shift+P`.
2. Run **Extensions: Install from VSIX...**.
3. Select `C:\dq\VSCode\dq-syntax-*.vsix`, using the actual filename and path.
4. Reload the VS Code window if prompted.

Open `C:\dq\examples` with **File > Open Folder**, then open `langdemo.dq`. Click the play button in the editor title bar or run **DQ: Run Current File** from the Command Palette. The extension saves the file and invokes `dq-run` with `-g -O0`.

The extension uses `dq-comp` as its language server. If you keep a portable installation outside `PATH`, set the paths explicitly in VS Code's `settings.json`:

```
{
    "dq.languageServerPath": "C:\\dq\\bin\\dq-comp.exe",
    "dq.runPath": "C:\\dq\\bin\\dq-run.exe"
}
```

Running a program with the play button does not start a debugger. For breakpoints and stepping, the Windows setup uses the **LLDB DAP** extension (`llvm-vs-code-extensions.lldb-dap`) with the bundled `lldb-dap.exe`. Its executable path can be configured in `settings.json`:

```
{
    "lldb-dap.executable-path": "C:\\dq\\toolchain\\llvm-mingw\\bin\\lldb-dap.exe"
}
```

Build debug executables with `-g -O0`. The VSCode Setup and Debugging chapter is the place for build-task and debugger launch configurations.

3.4.5 Add External Libraries with MSYS2

Packages such as the SQLite and MySQL bindings need external C libraries. A convenient source of Windows builds is MSYS2.

Install MSYS2, using `C:\msys64` for the paths below. Open **MSYS2 UCRT64** from the Start menu and update the installation:

```
pacman -Syu
```

If the update asks you to close the terminal, reopen **MSYS2 UCRT64** and run the command again to finish. Follow the MSYS2 update instructions for further details.

Install whichever client libraries your application needs:

```
pacman -S --needed mingw-w64-ucrt-x86_64-sqlite3
pacman -S --needed mingw-w64-ucrt-x86_64-libmariadbclient
```

The SQLite package provides the SQLite runtime and development libraries. The MariaDB client package provides libraries for the MySQL-compatible client interface. Installing a client library does not install a database server.

Use the **UCRT64** environment and packages beginning with `mingw-w64-ucrt-x86_64-` for this setup. Its libraries live under `ucrt64`. MSYS2's other environments use different toolchain or runtime combinations; see the MSYS2 environment overview.

3.4.5.1 Configure Library and DLL Search Paths

DQ needs to find the libraries during linking and their DLLs when the program runs. Add these directories to your user environment:

Variable	Directory	Purpose
`PATH`	`C:\dq\bin`	DQ tools and their runtime DLLs.
`PATH`	`C:\msys64\ucrt64\bin`	External libraries' runtime DLLs.
`LIBRARY_PATH`	`C:\msys64\ucrt64\lib`	External libraries used by the linker.

Use the same Windows environment-variable editor as for the DQ tools. Create the user variable `LIBRARY_PATH` if it does not exist. If it already exists, add the new directory to its semicolon-separated list while preserving the existing directories. Keep the DQ `bin` entry ahead of the MSYS2 entry in `PATH`.

To configure the current Command Prompt temporarily:

```
set "PATH=C:\dq\bin;C:\msys64\ucrt64\bin;%PATH%"
set "LIBRARY_PATH=C:\msys64\ucrt64\lib;%LIBRARY_PATH%"
```

Open a new terminal and restart VS Code after changing persistent settings. You can then run the SQLite example from the extracted DQ package directory:

```
cd /d C:\dq
dq-run stdpkg\sqlite3\examples\sqlite_test.dq
```

3.4.6 Run Applications on Another Computer

When distributing a compiled program, include the runtime DLLs it needs. For the standard Windows runtime, these normally include `libc++.dll` and `libunwind.dll` from the DQ package's `bin` directory. Include any additional DLLs required by the particular build, such as `libwinpthread-1.dll` when used.

Applications that use external libraries also need those libraries' runtime DLLs and their dependencies. For example, SQLite's DLL is `C:\msys64\ucrt64\bin\libsqlite3-0.dll`.

Place the required DLLs beside the application executable, preserving their filenames. You can inspect direct DLL imports with the bundled tool:

```
"C:\dq\toolchain\llvm-mingw\bin\llvm-readobj.exe" --coff-imports hello.exe
```

Static linking is available through `dq-comp --static`, when the required static libraries are available. It can reduce external DLL dependencies, but does not guarantee a completely standalone executable. Include the applicable license notices for libraries you distribute.

3.4.7 Troubleshooting

3.4.7.1 Windows Cannot Find `dq-comp`

Run `where dq-comp`. Check that your DQ `bin` directory is in `PATH` and that you opened a new terminal after editing the environment. If running from VS Code, restart the entire editor rather than only its integrated terminal.

3.4.7.2 The Linker Cannot Find SQLite

For an error such as `unable to find library -lsqlite3`, check:

```
echo %LIBRARY_PATH%
dir C:\msys64\ucrt64\lib\libsqlite3.dll.a
```

Install the UCRT64 SQLite package if the import library is missing. Adding `ucrt64\bin` to `PATH` supplies DLLs for execution; `LIBRARY_PATH` supplies libraries for linking.

3.4.7.3 A Runtime DLL Is Missing

Check `where libsqlite3-0.dll` for SQLite and your DQ `bin` directory for `libc++.dll` and `libunwind.dll`. Verify the `PATH` inherited by the program or place the required DLLs beside its executable.

3.4.7.4 A Library Has the Wrong Architecture or Format

Use native x86_64 Windows UCRT libraries with this compiler package. Linux/WSL libraries and MSYS libraries do not replace those native Windows libraries. Check that both the selected package and the search paths use `ucrt64`.

3.5. Compiler Usage and Options

Compile a source file or a project file with `dq-comp`. Use `dq-comp --help` to see the options accepted by your installed version.

3.5.1 `dq-comp`

Basic use:

```
dq-comp app.dq
./app
```

Useful options:

Option	Meaning
`-o <file>`	set output filename
`-c`	compile only; do not link
`--ifgen`	generate a standalone `.dqm_if` interface file
`--ifdump`	dump a standalone `.dqm_if` interface
`--no-use-sys`	disable implicit merged `sys` import
`--target=<name>`	select a compiler target
`--sysroot=<path>`	select the target system root used for linking
`--cpu-features=<features>`	append LLVM CPU features to the target defaults
`--targets`	list canonical targets and their LLVM/default settings
`--exceptions`, `--no-exceptions`	enable or disable DQ exception handling
`--dynstrings`, `--no-dynstrings`	enable or disable heap-backed dynamic strings
`--pkg_path <path>`	add a package search root; repeatable; alias: `--pkg-path`
`--pkg <name:directory>`	map a package directly; repeatable; takes priority over search roots
`--build <tag>`, `--build-tag <tag>`	select `.dqbuild/<tag>`
`--build-suffix <suffix>`	append to the selected build tag
`--build-root <path>`	select the local artifact build root
`--package-build-root <path>`	select a separate package artifact build root
`--lto[=full|off]`	control full LTO bitcode sidecars
`-Dname`	define a compile-time symbol as true
`-Dname=value`	define an integer or boolean symbol
`-O0` .. `-O3`, `-Os`, `-Oz`	optimization level
`-g`	generate debug info
`--static`	link statically; equivalent to `--linker-arg=-static`
`-v`, `-vv`, `-vvv`	increasing compiler verbosity
`-ir`	print generated LLVM IR

The default optimization level is currently `-O1`.

The `rtl/sys` prelude is implicitly merged for hosted and bare targets. Use `--no-use-sys` only for low-level bootstrap or RTL work that must opt out. Exception handling defaults to enabled on hosted targets and disabled on bare targets; when enabled, `#ifdef EXCEPTIONS` selects exception-aware source code. Runtime checks remain active when exception handling is disabled. Dynamic strings follow the same target defaults; when enabled, `#ifdef DYNSTRINGS` selects `str`-dependent source. With dynamic strings disabled, `strslice`, `rostr`, string literals, and bounded `embstr` remain available, but owning `str` operations are rejected at compile time.

3.5.2 Package Search Path

Bare imports such as `use print` or `use json` are resolved through package search roots. The compiler adds standard roots automatically, including:

```
/usr/lib/dq/stdpkg
<compiler-dir>/../lib/dq/stdpkg
<compiler-dir>/../stdpkg
/usr/lib/dq/packages
<compiler-dir>/../lib/dq/packages
~/.dq/packages
```

Additional roots can be appended with `--pkg_path`. Search uses the last matching package root first, so later `--pkg_path` entries can override earlier roots. Both `--pkg_path=path` and `--pkg_path path` are accepted.

Use `--pkg=board:../boards/adcif` (or `--pkg board:../boards/adcif`) to select a specific source directory for `board`. Explicit mappings override all search roots, command-line mappings override project mappings, and the last mapping for a name wins. A mapped directory must exist; missing files within it never fall back to another package. CLI paths are relative to the working directory. The equivalent project properties are `pkg_path` and `pkg`; see package lookup.

3.5.3 Linking Native Code

`--link` requests linking explicitly; `-c` produces an object without linking. `--linker-arg=<arg>` passes an argument to the native linker. For repeatable linker inputs, library directories, or a linker script, use the corresponding project properties.

`DQ_LINKER_DRIVER` selects the Clang driver used for linking. Match its LLVM major version to the compiler's LLVM version, particularly for full LTO builds. The full release packages supply a matching toolchain.

See Targets and Cross-Compilation for application targets, sysroots, and Windows cross-linking. The tool overview covers `dq-run`.

3.6. Project Files (`.dqproj`)

Some projects may require a large amount of compiler options to compile. In order to make the complex project compilation easier the DQ compiler supports project files.

The DQ project files are basically a set of compiler options in a more manageable way.

The project files use the `.dqproj` extension:

Example usage:

```
dq-comp application.dqproj
dq-comp -O0 -g -o build/application application.dqproj
```

(the dq-comp options provided in the command line override the options in the .dqproj file)

Unless `output` or `-o` selects another path, the executable is named after the project file and written beside it. Build artifacts are kept below the project directory's `.dqbuild` directory, independently of the `main` source location.

3.6.1 Project File Example

```
// application.dqproj

pkg_path = 'packages'
var BOARD_SUPPORT = PackagePath('board_support')

include '\{BOARD_SUPPORT}/project/stm32f746.dqproj'

main      = 'src/application.dq'
output    = 'build/\{PROJECT_NAME}.elf'
target    = 'arm_m7f-bare'
cpu_features = '+no-movt'
compiler_runtime = 'libgcc'
c_runtime = 'newlib-nano'
link      = true
optlevel  = 2
debuginfo = true
lto       = false

define BOARD_NUCLEO_F746
define CPU_CLOCK_HZ = 216000000

linkobject = '\{BOARD_SUPPORT}/lib/startup.o'
linkoption = '--gc-sections'
```

`arm_m7f-bare` selects the Cortex-M7 single-precision FPU and makes `float` 32-bit. Use `arm_m7fd-bare` for Cortex-M7 devices with a double-precision FPU; on that target `float` is 64-bit. The explicit `float32` and `float64` types do not change with the target.

Use `dq-comp --targets` for the canonical target list and detailed defaults. The additional hosted Linux targets are `armhf-linux`, `arm64-linux`, and `rv64g-linux`; select the target distribution or SDK root with `sysroot` when linking for one of them. The additional cross targets are `wasm32-wasi`, `wasm32-bare`, and `rv32imac-bare`. WASI exceptions are unsupported, so a project selecting `wasm32-wasi` must not set `exceptions = true`.

An included fragment might contain the shared target and linker configuration:

```
// board_support/project/stm32f746.dqproj

linkerpath = '\{THIS_DIR}/../lib'
linkscript = '\{THIS_DIR}/../ld/stm32f746.ld'
```

3.6.2 Lexical Rules

The format uses DQ-like identifiers, strings, comments, and statement terminators. Property and keyword names are case-sensitive.

An identifier starts with an ASCII letter or `_` and may continue with ASCII letters, decimal digits, or `_`:

```
FEATURE
CPU_CLOCK_HZ
board2
```

Strings may use single or double quotes. They must end on the same physical line on which they begin.

```
'src/main.dq'
"arm_m7f-bare"
```

The recognized string escapes are:

Escape	Result
`\n`	newline character
`\r`	carriage return
`\t`	tab
`\\`	backslash
`\"`	double quote
`\'`	single quote

An unrecognized escape is preserved as a backslash followed by the escaped character.

Both DQ comment forms are supported:

```
// line comment

/* block
   comment */
```

An unterminated string or block comment is an error.

Statements normally end at a newline. A semicolon is optional at the end of a statement and is required between multiple statements on the same line:

```
main = 'main.dq'
output = 'app'

main = 'main.dq'; output = 'app'
```

The parts of one statement, including `=`, its value, and the arguments of `PackagePath()`, may not continue onto another line.

3.6.3 Statements

A project consists of property assignments and three special statements: `define`, `var`, and `include`.

```
property = value
define NAME
define NAME = value
var NAME = value
include 'file.dqproj'
```

Unknown properties, invalid value types, trailing tokens, and duplicate single-value properties are errors.

3.6.3.1 Properties

Property	Value	Repetition	Meaning
`main`	path string	once, required	DQ source file compiled as the project entry module.
`output`	path string	once, optional	Final executable or object filename. Normal compiler output defaults apply when omitted.
`target`	string	once, optional	Compiler target name, using the same names accepted by `--target`.
`build_tag`	nonempty string	once, optional	Base name of the `.dqbuild` artifact directory; defaults to the selected target name.
`build_suffix`	nonempty string	once, optional	Append `-<suffix>` to the selected build tag.
`sysroot`	path string	once, optional	Target system root passed to the Clang linker driver for hosted cross builds.
`cpu_features`	string	once, optional	Append LLVM CPU feature flags to the target defaults, separated by commas.
`pkg_path`	path string	repeatable	Add a DQ package search root (`packagepath` is a compatibility alias).
`pkg`	`name:directory` string	repeatable	Map one package directly to a source directory; overrides all search roots.
`link`	boolean	once, optional	Force linking when `true`; compile only when `false`.
`static`	boolean	once, optional	Pass `-static` to the linker when `true`, equivalent to `--static` or `--linker-arg=-static`. Defaults to `false`, which adds no linker argument.
`exceptions`	boolean	once, optional	Enable or disable DQ exception handling. Defaults to enabled on hosted targets and disabled on bare targets; `wasm32-wasi` always disables and rejects enabling exceptions.
`dynstrings`	boolean	once, optional	Enable or disable heap-backed dynamic strings. Defaults to enabled on hosted targets and disabled on bare targets.
`compiler_runtime`	string	once, optional	Bare-target compiler support runtime: `libgcc` (default) or `none`.
`c_runtime`	string	once, optional	Bare-target C runtime: `newlib-nano` (default) or `none`.
`linkobject`	path string	repeatable	Add an existing object or other positional linker input.
`linkerpath`	path string	repeatable	Pass `--library-path=<path>` to the linker.
`linkoption`	string	repeatable	Pass one argument directly to the linker.
`linkscript`	path string	once, optional	Pass `--script=<path>` to the linker.
`debuginfo`	boolean	once, optional	Enable or disable debug information.
`optlevel`	integer or string `0` through `3`, or string `'s'` or `'z'`	once, optional	Set the optimization level. Numeric levels may be quoted or unquoted; `'s'` and `'z'` optimize for size.
`lto`	boolean	once, optional	

Property	Value	Repetition	Meaning
			Enable full LTO when `true`; disable LTO when `false`.

`main` is required only after the top-level file and all its includes have been assembled. An included fragment therefore does not need its own `main`. `output` is always optional.

For example:

```
build_tag = 'stm32f746'
build_suffix = 'debug'
```

This selects `.dqbuild/stm32f746-debug/`. Either property may be used alone, and both support project-variable expansion such as `\{PROJECT_NAME}`.

The `cpu`, `abi`, and `floatabi` property names are reserved for future target tuning. They are currently rejected as unsupported. They are not silently ignored.

3.6.3.2 Link Mode

Without a `link` property, the compiler retains its automatic link mode:

- A hosted target links a module that contains `*Main`.
- A bare target produces an object by default.

`link = true` is equivalent to explicitly selecting `--link`. It can force a bare target to link an ELF image. `link = false` is equivalent to `-c` and always requests compile-only output.

3.6.3.3 Exception Handling

Exception handling is enabled by default for hosted targets and disabled by default for bare targets. Projects can override the target default explicitly:

```
exceptions = false
```

When enabled, the compiler defines `EXCEPTIONS` in the `@def` scope, so source and RTL code can select exception-dependent code with `#ifdef EXCEPTIONS`. Disabling exceptions removes native unwind generation and rejects `try`, `except`, `finally`, and `raise`; runtime checks remain enabled and report through the `RuntimeError` hook.

3.6.3.4 Dynamic Strings

Dynamic strings are enabled by default for hosted targets and disabled by default for bare targets. Projects can override the target default explicitly:

```
dynstrings = true
```

When enabled, the compiler defines `DYNSTRINGS` in the `@def` scope. When disabled, heap-owning `str` declarations, conversions, concatenation, and methods are rejected with `DynStringsDisabled`. Non-owning and bounded text facilities—text literals, `rostr`, `strslice`, and bounded `embstr`—remain available.

3.6.3.5 Bare-Metal Runtimes

For supported bare-metal targets, the compiler selects bundled libraries that match the target CPU and floating-point ABI. The defaults are equivalent to:

```
compiler_runtime = 'libgcc'
c_runtime = 'newlib-nano'
```

`compiler_runtime` supplies compiler-generated ABI helper routines and, when exceptions are enabled, the C++ exception ABI support and target unwinder. `c_runtime` supplies Newlib Nano and its no-system fallback stubs, and makes the matching bundled math library available to `#linklib('m')`. Static archive members are included only when referenced. Board or application implementations override the fallback stubs and must provide functional system calls where the application requires them.

Either property may be set to `none` when the application supplies that part of the runtime explicitly. These properties are rejected for hosted targets, whose runtime libraries continue to be selected by the system linker driver.

3.6.3.6 Defines

`define` creates a symbol in the compiler's `@def` scope:

```
define FEATURE
define LOGGING = true
define TRACE = false
define BUFFER_SIZE = 4096
define OFFSET = -16
```

A define without a value has boolean value `true`. Explicit values are limited to `true`, `false`, or a signed 64-bit decimal integer. Strings, hexadecimal integers, expressions, and variables are not accepted as define values.

Define names are identifiers and must be unique throughout the assembled project, including all included files. A command-line `-DNAME` or `-DNAME=value` replaces the project define with the same name.

Project defines are inherited by imported child modules.

3.6.4 Paths

The following properties contain paths:

- `main`
- `output`
- `pkg_path`
- The directory part of `pkg`
- `linkobject`
- `linkerpath`
- `linkscript`
- `include`

After variable expansion, a relative path is resolved against the directory of the file containing that statement. The result is converted to an absolute, lexically normalized path. This rule applies independently to every included file.

For example, in `project/boards/board.dqproj`:

```
linkscript = '../ld/board.ld'
```

the path is relative to `project/boards`, not to the process working directory or the top-level project directory.

The project parser requires included files, mapped `pkg` directories, and `PackagePath()` results to exist. It does not check the existence of `main`, `output`, `pkg_path`, `linkobject`, `linkerpath`, or `linkscript`; the compiler, filesystem, or linker reports problems with those paths when they are used.

3.6.5 Variables

A variable stores a path for later substitution. Its value must be either a quoted path string or the result of `PackagePath()`:

```
var SOURCE_DIR = 'src'
var SDK = PackagePath('sdk')

main = '\{SOURCE_DIR}/main.dq'
include '\{SDK}/project/common.dqproj'
```

A relative quoted value is resolved against the directory of the file that declares the variable. Consequently, it keeps the same meaning when it is used later from a different included file.

Variable references use only the explicit `\{NAME}` form. Expansion is performed inside all quoted project values, including path properties, `target`, `linkoption`, include paths, and `PackagePath()` arguments.

Variable names are case-sensitive, must be declared before use, and are shared through includes. A variable cannot be declared more than once. User variables also cannot redefine a built-in variable:

Variable	Meaning
`\{PROJECT_NAME}`	Name of the top level `.dqproj` file, without extension
`\{PROJECT_DIR}`	Directory containing the top-level `.dqproj` file.
`\{THIS_DIR}`	Directory containing the file with the current statement.

The project language does not expand environment variables, `~`, `$NAME`, or `%NAME%`. Dollar signs have no special meaning. Write `\\{NAME}` to insert literal `\{NAME}` text without expanding a variable.

3.6.6 Package Lookup

`PackagePath()` locates a package using the same package-root resolver as a DQ `use` statement:

```
var SDK = PackagePath('sdk')
```

The argument must expand to one identifier. If `/opt/dq/packages/sdk` is the selected package, the function returns the absolute normalized path `/opt/dq/packages/sdk`, not `/opt/dq/packages`.

Packages are selected using the following precedence, highest first:

1. Command-line `--pkg` mappings for that package name.
2. Project `pkg` mappings already encountered for that package name.
3. Command-line `--pkg_path` roots.
4. Project `pkg_path` roots already encountered.
5. Compiler default package roots.

Within each level, the last matching entry wins. Explicit mappings always take priority over search roots, including a project mapping over a command-line root.

Command-line package roots and mappings are collected before the project is evaluated, so they are available to every `PackagePath()` call regardless of command-line position. Project `pkg_path` and `pkg` settings become available only after their statements are evaluated. Includes follow the same textual ordering rule.

For example:

```
pkg_path = '../dqpkg'
pkg = 'board:../boards/adcif'
var BOARD = PackagePath('board')
```

This binds `board` to `../boards/adcif`, even if a search root contains another `board` directory. `use board` selects `../boards/adcif/board.dq`, while `use board/pins` selects `../boards/adcif/pins.dq`. Package-qualified includes, `^/` paths, and language-server navigation use the same mapping.

The mapping is split at the first colon. The name must be one identifier; variable expansion applies to the whole string, and only the directory part is resolved relative to the file containing the statement. Spaces and further colons are part of the directory, including Windows drive prefixes.

A missing or inaccessible mapped directory is an error when the declaration is processed. A missing module or include inside a mapped package does not fall back to search roots. A missing `PackagePath()` package is also a project error.

The old project spelling `packagepath` remains an alias for `pkg_path`. On the command line, `--pkg_path` and `--pkg` accept either `--option=value` or `--option value`; `--pkg-path` remains an alias for `--pkg_path`. CLI directory paths are relative to the process working directory.

3.6.7 Includes

An include inserts another `.dqproj` file at the point of the statement:

```
include 'common.dqproj'
include '\{SDK}/project/board.dqproj'
```

The path is expanded and resolved relative to the including file. Included files must also use the `.dqproj` extension.

Includes behave like textual insertion with file-local path anchoring:

- Variables declared before an include are visible inside it.
- Variables declared by an included file remain visible after it returns.
- `pkg_path` and `pkg` statements affect later `PackagePath()` calls.
- Single-property, variable, and define duplicate checks span all files.
- Repeatable property ordering follows statement order across includes.

The same file may be included again after its earlier inclusion has completed. Normal duplicate rules still apply to the repeated statements. Including a file that is already active is a cycle and is rejected with the canonical include chain.

3.6.8 Linker Argument Ordering

`linkerpath`, `linkscript`, `linkoption`, and `static = true` produce linker arguments in their assembled project statement order. Included statements retain their position in that order. Command-line `--linker-arg=<arg>` and `--static` entries follow all project linker arguments.

The compiler constructs linker inputs in this order:

1. The object generated for `main`.
2. Objects generated for imported DQ modules.
3. Project `linkobject` inputs, in statement order.
4. `-o` and the final output filename.
5. Libraries requested by DQ modules.

Each project or command-line linker argument is passed through the compiler driver as one linker argument before those positional inputs. `-static` is passed directly to the driver so it also selects static startup files and runtime libraries.

Project parsing does not verify that linker objects, library paths, or linker scripts exist. Such errors are reported by the linker.

3.6.9 Command-Line Precedence

The project is parsed before normal command-line option evaluation and before target-dependent compiler initialization. Project values act as defaults; explicit command-line selections are then applied on top.

Setting	Precedence
Target	`--target` overrides `target`; otherwise the project target overrides the host default.
Build tag	`--build` or its alias `--build-tag` overrides `build_tag`; otherwise the selected target name is used when `build_tag` is absent.
Build suffix	`--build-suffix` overrides `build_suffix`. The effective suffix is appended once to the effective build tag, including a tag selected by `--build`.
CPU features	`--cpu-features=` overrides `cpu_features`; the effective value is appended to the selected target defaults.
Output	`-o` overrides `output`; otherwise normal output defaults apply when the property is absent.
Optimization	`-O0` through `-O3`, `-Os`, and `-Oz` override `optlevel`.
Debug information	`-g` enables debug information even when `debuginfo = false`. There is currently no command-line option to force it off.
LTO	`--lto` or `--lto=full|off` overrides `lto`.
Exceptions	`--exceptions` or `--no-exceptions` overrides `exceptions`; the last command-line flag wins.
Dynamic strings	`--dynstrings` or `--no-dynstrings` overrides `dynstrings`; the last command-line flag wins.
Link mode	The first explicit `-c` or `--link` replaces the project mode. Conflicting subsequent command-line link modes are errors.
Defines	The first command-line definition of a name removes the project definition of that name.
Package roots	Command-line roots have higher lookup precedence than project and default roots.
Linker arguments	Project arguments precede command-line `--linker-arg` and `--static` entries.

The project target is selected early enough to initialize target-dependent built-in types correctly. Imported child modules inherit the effective target, optimization level, debug and LTO settings, defines, build settings, and package roots through compiler arguments. Child compilations receive the resolved `.dq` source file and do not parse the project again.

3.6.10 Errors and Diagnostics

Project evaluation stops at the first error. Diagnostics identify the file, line, column, diagnostic identifier, and message:

```
/work/app/application.dqproj(12,8) ERROR(ProjectValue): optlevel must be 0 through 3, 's', or 'z'
```

Project errors include, among others:

- unreadable project or include files;
- wrong project extensions;
- malformed strings, comments, or statements;
- unknown, duplicate, or unsupported properties;
- invalid property types or values;
- duplicate or unknown variables;
- duplicate project defines;
- missing packages;
- include cycles;

- an assembled project without `main`.

3.6.11 Compact Grammar

The following grammar is descriptive. `newline` or `;` terminates a statement, and the final statement may end at end-of-file.

```
project       := { terminator | statement terminator } [ statement ] EOF

statement     := property '=' property-value
               | 'define' identifier [ '=' define-value ]
               | 'var' identifier '=' variable-value
               | 'include' string

property      := 'main' | 'output' | 'target' | 'build_tag' | 'build_suffix'
               | 'pkg_path' | 'packagepath' | 'pkg'
               | 'link' | 'static' | 'linkobject' | 'linkerpath' | 'linkoption'
               | 'linkscript' | 'debuginfo' | 'optlevel' | 'lto'
               | 'exceptions' | 'dynstrings' | 'cpu_features'

property-value := string | boolean | signed-decimal-integer
variable-value := string | 'PackagePath' '(' string ')'
define-value   := boolean | signed-decimal-integer
boolean        := 'true' | 'false'
terminator     := newline | ';'
```

Whitespace and comments may occur between tokens as long as they do not move the remainder of the current statement onto another line.

3.7. Build Process and Module Artifacts

3.7.1 Artifact Locations

DQ packages and the RTL are source-first. The compiler writes generated module artifacts under `.dqbuild`.

```
.dqbuild/<build-tag>/
  local/
    app.o
    app.dqm_if
  pkg/
    print/
      print.o
      print.dqm_if
    rtl/
      rtl_linux.o
      rtl_linux.dqm_if
```

Each module has a normal native `.o` object and a standalone `.dqm_if` interface. Both are generated artifacts, not source package contents.

Local imports outside the build root are stored under `.dqbuild/<build-tag>/external/`, using their absolute source path below that directory (and a drive identifier on Windows). This keeps parent-relative imports inside the build cache and distinguishes sources with identical filenames.

By default the build root is the directory containing the main `.dq` file. The default build tag is `<target-arch>-<target-rtl>`, for example `x86_64-linux`.

The build tag is only a namespace. A tag named `debug` does not imply `-g`, and a tag named `release` does not imply `-O3`.

By default, local and package artifacts use the same build root. With `--package-build-root <path>`, package and RTL artifacts are instead written to `<path>/.dqbuild/<build-tag>/pkg`, while local artifacts remain under the normal build root. This allows several projects with the same build context to share compiled package modules.

DQ compilation uses paired `.o` and `.dqm_if` artifacts.

Artifact	Meaning
`.o`	native compiled module object file
`.dqm_if`	serialized public module interface and freshness metadata

Both files normally live under `.dqbuild`. The object is a regular linker input and contains no DQ-specific interface section. The compiler always reads semantic module data from the standalone `.dqm_if`.

3.7.2 Why Interfaces Exist

When a module imports another module, the compiler needs that imported module's public declarations: exported functions, objects, types, constants, properties, and imported or reexported interface symbols. Loading a compact interface is faster and simpler than reparsing implementation code for every import.

3.7.3 Interface Payload

The `.dqm_if` payload is a compiler-owned binary format designed for fast loading, strict validation, regeneration, and versioning. It records source dependencies with their paths, sizes, and modification times. Interfaces emitted by a full compile also identify the matching object by its size and modification time.

Reexports are flattened into the facade interface. The payload contains:

- the facade module's own public declarations;
- selected reexported declarations, grouped by their canonical origin module;

- facade export references that map visible names to the canonical declarations;
- non-exported declarations required to describe exported types and signatures.

The last category is the referenced-type closure. It includes aliases, bases, fields, methods, properties, function parameters and results, and nested pointer, array, function-reference, and object-reference types. Closure-only declarations remain inaccessible through the facade.

Named type references in embedded declarations identify both their origin module and original name. Consequently a type reached through a facade and the same type reached through a later direct import share canonical ownership, linkage identity, and one in-memory representation.

Freshness metadata is flattened as well. A facade records the source metadata and canonical module-to-source resolution for its complete public-use dependency graph. Loading the facade therefore reads only its own `.dqm_if`; child interface files are not opened merely to load reexported declarations. Child source files are still checked with filesystem metadata, and a facade is regenerated if a source changes or package-path resolution selects a different source.

The binary format is versioned as a compiler cache. Incompatible interfaces are rejected and regenerated rather than read through an older reexport layout.

It is not a source format, is not intended for hand editing, and is not a stable ABI contract between unrelated compiler versions.

3.7.4 Inspecting Interfaces

Use `--ifdump` to inspect a standalone `.dqm_if`.

```
dq-comp --ifdump .dqbuild/x86_64-linux/local/app.dqm_if
```

For flattened facades, the dump lists facade export mappings and origin groups. Declarations used only for the referenced-type closure are marked `closure`.

Use `--ifgen` when only the public interface is needed.

```
dq-comp --ifgen module.dq
```

Normal builds do not require checking in `.o` or `.dqm_if` files.

3.7.5 Cleaning

Clean `.dqbuild/<tag>` when the compiler version, target, ABI-relevant options, package sources, or important build defines change.

```
rm -rf .dqbuild/x86_64-linux
```

Rebuild after changing compiler versions or target settings if an old cache is in doubt. Keep `.dqbuild` out of source control; packages are distributed as source, and generated interfaces are specific to a compiler/build context.

The compiler also tracks textual include and `#srcdep` inputs. For source-level visibility and reexport rules, see Modules and Packages. Contributor details are in Module and Interface Internals.

3.8. Targets and Cross-Compilation

A target describes the machine and runtime on which your **DQ application** will run. Selecting a foreign application target does not require rebuilding `dq-comp`. To build a compiler executable that itself runs on another machine, see Compiler Cross-Builds.

3.8.1 Compiler Targets

Run `dq-comp --targets` to print the canonical target names together with their architecture, platform, LLVM triple, CPU/features, feature defaults, and default link mode. In addition to the current host and Cortex-M targets, the compiler supports:

Target	Behavior
`x64-windows`	Windows x64 GNU/UCRT executable; also accepts `x86_64-win`, `x64-win`, and `x86_64-windows`
`armhf-linux`	ARMv7-A hard-float Linux executable
`arm64-linux`	AArch64 Linux executable
`rv64g-linux`	RV64G/LP64D Linux executable
`wasm32-wasi`	Hosted WASI command module; links automatically and defaults to a `.wasm` output
`wasm32-bare`	Bare WebAssembly object generation; compile-only by default
`rv32imac-bare`	Bare RV32IMAC ELF object generation; compile-only by default

`wasm32-wasi` requires Clang/LLD and a WASI sysroot containing libc at link time. DQ exceptions are not supported on this target: `--exceptions` and `exceptions = true` are rejected, while dynamic strings remain enabled by default. The resulting command module uses WASI libc startup and can be run by a WASI runtime such as Wasmtime.

The foreign Linux targets require a target sysroot at link time. Select it with `--sysroot=/path/to/sysroot` or the `sysroot` project property. DQ passes this to its Clang linker driver as `--sysroot`; the sysroot must come from the target distribution or SDK and match the deployed C library and development packages.

For example, an ARM64 Debian sysroot is selected per build:

```
dq-comp --target=arm64-linux --sysroot=/path/to/debian-arm64-sysroot app.dq
```

The two bare targets do not bundle startup code, linker scripts, libc, or compiler-runtime libraries. They emit objects without linking unless `--link` is requested. A forced bare link must supply the appropriate objects and linker arguments and normally sets `compiler_runtime = 'none'` and `c_runtime = 'none'`.

Target selection defines architecture and ABI markers as applicable: 32-bit x86 uses `X86_FAMILY` and `X86`, while 64-bit x86 uses `X86_FAMILY` and `X64`. Arm targets use `ARM` plus `ARM_M` or `ARM_A`; the hosted hard-float target also defines `ARMHF`. AArch64 defines `ARM_A`, `ARM64`, and `AARCH64`. RISC-V targets define `RISCV` plus `RV32` or `RV64`. Bare targets also define `BARE`, and `WASM` / `WASI`, `TARGET_32BIT` / `TARGET_64BIT`, `EXCEPTIONS`, `DYNSTRINGS`, and `HARDFLOAT` / `HARDFLOAT32` / `HARDFLOAT64` continue to reflect the selected target and feature settings.

3.8.1.1 Windows cross-compilation from Linux

When built with LLVM's X86 backend, the Linux compiler can produce Windows x64 executables using a Linux-hosted llvm-mingw UCRT toolchain. With the toolchain extracted under `sysroots/llvm-mingw-x86_64`, source the setup script in Bash before compiling and running:

```
source tools/dqcrosswinsetup.sh
build/dq-comp --target=x64-windows app.dq
wine app.exe
```

The script detects the repository root from its own location, sets `DQ_MINGW_ROOT`, and uses `winepath` to set `WINEPATH` to the toolchain's target runtime DLL directory. Wine loads the DLLs directly from that directory, so you do not need to copy them beside

each executable. The script also sets `MESA_VK_IGNORE_CONFORMANCE_WARNING=1` to suppress Mesa's RADV Vulkan conformance warning. Wine and `winepath` must be installed.

You can source the script by its absolute path from any working directory. For example, add this line to `~/.bashrc` to configure new Bash terminals automatically (adjust the path to your checkout):

```
source /path/to/dq-lang/tools/dqcrosswinsetup.sh
```

Source the script in the current terminal as well to apply the settings immediately. Running it as a separate process does not configure the current shell.

Windows outputs default to `.exe`. An explicit `-o` filename is used exactly as given, including filenames without an extension. Object generation with `-c` does not require the Windows libraries or Wine.

Set `DQ_MINGW_ROOT` to the extracted llvm-mingw root when it is installed elsewhere:

```
DQ_MINGW_ROOT=/path/to/llvm-mingw build/dq-comp --target=x64-windows app.dq
```

The linker search order is `DQ_LINKER_DRIVER`, the cross driver under `DQ_MINGW_ROOT/bin`, compiler-relative `toolchain/llvm-mingw`, `llvm-mingw`, and `sysroots/llvm-mingw-x86_64` installations, then `x86_64-w64-mingw32-clang++` on `PATH`. An invalid explicit `DQ_MINGW_ROOT` fails when linking. Native Windows compiler installations retain their Windows-hosted driver discovery. `--sysroot` is still passed through when set.

The toolchain must provide Clang/LLD, startup objects, UCRT and Windows import libraries, libc++, and libunwind. Full LTO also requires a toolchain with the same LLVM major version as DQ. Windows full-LTO links use LLD's default optimization pipeline rather than the ELF-specific O1 inlining pipeline.

As an alternative to setting `WINEPATH`, copy the toolchain's target runtime DLLs beside the executable: `libc++.dll`, `libunwind.dll`, and `libwinpthread-1.dll` when provided. These are under `x86_64-w64-mingw32/bin` in llvm-mingw.

The opt-in compatibility suite copies the runtime DLLs automatically:

```
make test-windows
# Custom installation:
DQ_MINGW_ROOT=/path/to/llvm-mingw make test-windows
# Direct invocation with explicit tool paths:
python3 tools/test-windows.py --compiler build/dq-comp \
    --mingw-root /path/to/llvm-mingw --wine /path/to/wine
```

It requires Linux, Python 3, llvm-mingw, and usable 64-bit Wine. Missing prerequisites fail the requested run; they are not silently skipped. Windows fixtures live in `autotest/cross-windows`; the suite also reuses the portable thread/channel autotests and the threads examples. It checks native thread lifecycle and handle cleanup, atomics, event deadlines, channel capacity and waits, producer/consumer stress, and work distribution at O0 and O3 with full LTO. Normal `make test`, `make autotest`, and `make check` runs do not invoke the Windows suite or require these dependencies. Executables, caches, runtime DLLs, command logs, and a dedicated Wine prefix are stored under `build/test-windows`. The suite leaves the user's Wine prefix untouched.

3.8.2 Bare-Metal Projects

Bare targets need board-specific startup objects, a linker script, and suitable runtime libraries when linking a complete image. Put those settings in a project file. The bundled support libraries cover the supported CPU/ABI combinations; they do not supply a board's reset handler or hardware initialization.

3.8.3 WebAssembly Examples

A WASI command module uses a hosted C library:

```
dq-comp --target=wasm32-wasi --sysroot=/path/to/wasi-sysroot app.dq
wasmtime app.wasm
```

For a browser or another custom embedding, `wasm32-bare` produces an object by default. Supply the embedding's link options and imports explicitly. See C Interoperability for import declarations, and `examples/wasm`, `examples/wasm_webgl`, and `examples/wasm_gpu` in the repository for integration examples.

3.9. VSCode Setup and Debugging

Install the DQ tools using the Linux or Windows environment chapter first. Confirm that `dq-comp --version` and `dq-run --help` work in the terminal from which you start VS Code.

3.9.1 DQ Extension

Install the bundled `VSCode/dq-syntax-*.vsix` with **Extensions: Install from VSIX...** in the Command Palette, then reload VS Code. In a source checkout, the extension files are under `tools/vscode-dq`.

The extension supplies syntax highlighting, compiler diagnostics, document symbols, block completion, the `$dq` problem matcher, and **DQ: Run Current File**. The run command uses `dq-run` with `-g -O0`. It does not start an interactive debugger.

Both tools must be on the editor's `PATH`. Alternatively set `dq.languageServerPath` and `dq.runPath` to their full paths in `settings.json`. On Linux, starting VS Code with `code .` from a configured terminal lets it inherit that terminal's environment. Restart the editor after changing persistent environment variables.

3.9.2 Debugging on Linux

Install GDB and the Microsoft C/C++ extension (`ms-vscode.cpptools`). Open the program directory as a VS Code workspace. The following configuration builds and debugs the active `.dq` file; keep that file active when starting the task.

Create `.vscode/tasks.json`:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "DQ: build current file",
            "type": "shell",
            "command": "dq-comp",
            "args": [
                "-g", "-O0", "${file}",
                "-o", "${fileDirname}/${fileBasenameNoExtension}"
            ],
            "options": {"cwd": "${workspaceFolder}"},
            "group": "build",
            "problemMatcher": "$dq"
        }
    ]
}
```

Create `.vscode/launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "DQ: Debug current program",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}",
            "args": [],
            "cwd": "${fileDirname}",
            "stopAtEntry": false,
            "externalConsole": false,
            "MIMode": "gdb",
            "preLaunchTask": "DQ: build current file"
        }
    ]
}
```

Set a breakpoint and press `F5`. Add program arguments to the launch configuration's `args` array. For a `.dqproj` application, replace the active-file build argument with the project path, and set `program` to the output executable. The working directory controls relative file access at runtime.

You can also debug outside the editor:

```
dq-comp -g -O0 app.dq
gdb ./app
```

The repository contains optional GDB helpers in `tools/gdb/dq_gdbinit.py` and `tools/gdb/dq_printers.py` for inspecting DQ values.

3.9.3 Debugging on Windows

DQ programs can be source-debugged with the `lldb-dap` debugger included in the Windows DQ package. The compiler must generate debug information, so the debug build uses `-g -O0`.

3.9.3.1 Install the extensions

Install **LLDB DAP** (`llvm-vs-code-extensions.lldb-dap`) from the VS Code Marketplace and reload VS Code.

For DQ syntax support and the `$dq` build problem matcher, also install the bundled `VSCode\dq-syntax-0.1.4.vsix` using **Extensions: Install from VSIX...**.

The Microsoft C/C++ extension may remain installed, but the configuration below does not use its `cppdbg` adapter. On Windows, configuring `cppdbg` with `lldb-mi.exe` can hang in `WindowsDebugLauncher.exe`.

Open the DQ project directory as the VS Code workspace. The project may be in a different directory from the DQ installation.

3.9.3.2 Configure the target

Add the DQ compiler's bin directory (e.g. `c:\\prg\\dq\\bin`) to the system path, so the `dq-comp.exe` or `dq-run` exe are accessible from command line without full path specification.

Add these values to `.vscode/settings.json`:

```
{
    "dq.debugTarget": "examples/langdemo",
    "dq.debugCwd": "examples",

    "lldb-dap.executable-path": "C:\\prg\\dq\\toolchain\\llvm-mingw\\bin\\lldb-dap.exe"
}
```

(`dq.debugTarget` has no `.dq` extension!)

For example, to debug the NanoNet echo client, use:

```
{
    "dq.debugTarget": "stdpkg/nanonet/examples/binecho_client",
    "dq.debugCwd": "stdpkg/nanonet/examples"
}
```

3.9.3.3 Add the build task

Create `.vscode/tasks.json`:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "dq: build configured target",
            "type": "shell",
            "command": "dq-comp.exe",
            "args": [
                "-g",
                "-O0",
                "${config:dq.debugTarget}.dq",
                "-o",
                "${config:dq.debugTarget}.exe"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "group": "build",
            "problemMatcher": "$dq"
        }
    ]
}
```

3.9.3.4 Add the launch configuration

Create `.vscode/launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "DQ: Debug configured target (LLDB-DAP)",
            "type": "lldb-dap",
            "request": "launch",
            "program": "${workspaceFolder}\\${config:dq.debugTarget}.exe",
            "args": [],
            "cwd": "${workspaceFolder}\\${config:dq.debugCwd}",
            "stopOnEntry": false,
            "console": "internalConsole",
            "preLaunchTask": "dq: build configured target"
        }
    ]
}
```

3.9.3.5 Start debugging

1. Open the `.dq` source file and set a breakpoint.
2. Open **Run and Debug** (`Ctrl+Shift+D`).
3. Select **DQ: Debug configured target (LLDB-DAP)**.
4. Press `F5`.

VS Code builds the selected target first, then starts it under LLDB. Command-line arguments can be added to the `args` array in `launch.json`.

3.9.3.6 Troubleshooting

- If VS Code does not recognize the `lldb-dap` debug type, verify that the **LLDB DAP** extension is installed and reload the window.
- If startup hangs while mentioning `WindowsDebugLauncher.exe` and `lldb-mi.exe`, VS Code is using a `cppdbg` configuration. Select the `lldb-dap` configuration above instead.
- If a breakpoint is not resolved, confirm the build task still contains both `-g` and `-O0`, then rebuild the target.
- If the executable or a runtime DLL cannot be found, check the target path and the DQ installation on `PATH`. Keep its toolchain directories together.
- If the program reads relative files, set `dq.debugCwd` to the directory it normally runs from.

3.9.4 Diagnostics and Build Configuration

If diagnostics use the wrong compiler, check `dq.languageServerPath`. If the run command cannot find `dq-run`, check `dq.runPath` and the editor environment. The syntax extension and a debugger extension serve different purposes; install both when using interactive breakpoints.

A breakpoint requires the executable built from the source currently open. Use `-g -O0`, confirm the output and launch paths agree, and rebuild. Program arguments and working directories belong to the launch configuration; package roots and compiler flags belong to the build task or project file.

4. Standard Packages

The runtime and standard packages ship as DQ source. Begin with the Library Overview for imports and package discovery.

- `sys`: implicit runtime services, memory, and process arguments.
- `print`: console output.
- `file`: files, directories, and paths.
- `json`: JSON trees, parsing, and serialization.
- `threads`: native threads, atomics, and channels.
- `time`: timestamps, elapsed time, date formatting, and sleeps.
- `nanonet`: sockets and HTTP servers.
- `shell`: child processes, captured output, and environment variables.
- Additional Standard Packages: math, parsing, database bindings, and lower-level platform interfaces.

4.1. Library Overview

DQ provides runtime services and source packages. The compiler builds imported package modules for the selected target and stores the results in `.dqbuild`. Installing a package does not require prebuilding it for every application.

4.1.1 Runtime and Explicit Imports

The implicit `sys` interface supplies basic runtime helpers. Other packages are imported explicitly:

```
use print

func *Main():
    var message : str = Format("{} items", [3])
    PrintLn(message)
endfunc
```

`print` writes to standard output. `Format` and the string helpers are available through the implicit `sys` runtime interface, so they need no explicit import.

Need	Documentation
Process arguments, memory blocks, runtime helpers	`sys`
Console output	`print`
Formatting and text helpers	String Functions, Text Formatting
Files, directories, paths	`file`
JSON trees	`json`
Native threads and channels	`threads`
Timestamps, elapsed time, date formatting, sleeps	`time`
Event-driven sockets and HTTP	`nanonet`
Child processes, captured output, environment variables	`shell`
Math, parsing, databases, C bindings	Additional Standard Packages

4.1.2 Package Discovery and Platform Dependencies

A bare import is resolved through the compiler's package search roots. Add project packages with `--pkg_path` or `pkg_path` in a `.dqproj` file. Later roots take precedence. See Modules and Packages for identity and lookup rules.

Packages can depend on native libraries and platform APIs. For example, `threads` has hosted Linux and Windows implementations; database bindings need the corresponding client libraries. A package's presence in `stdpkg` does not imply that it is available on a bare-metal or WebAssembly target.

The package chapters include the maintained package-local API manuals, so the website, PDF book, and standalone package documentation use the same source.

4.2. `sys`

`rtl/sys` is implicitly merged into ordinary DQ modules. You can call its public helpers without writing `use sys`. `--no-use-sys` opts out for low-level modules.

Reexported module	Services
`rtl/rtlint`	Runtime intrinsics, type helpers, runtime error hook.
`rtl/mem`	Allocation and byte-block operations.
`rtl/strfunc`	Runtime string operations.
`rtl/exception`	`Exception` and exception helpers, when exceptions are enabled.
`rtl/prgargs`	Arguments and `exit_code`, on hosted targets.

String methods, `anyvalue`, and formatting are documented with their language chapters: String Functions, `anyvalue`, and Text Formatting.

4.2.1 Arguments and Process Status

```
func ArgCount() -> int
func ArgStr(i : int) -> ^char
func Exit(status : int)
func Errno() -> int32
```

On hosted targets, `ArgCount()` includes the program name at index zero. `ArgStr(i)` returns a borrowed C string or `nil` for an invalid index. Read the string before the process ends; do not free the pointer.

Set `exit_code` and return normally from `*Main` to select a process status. `Exit(status)` terminates immediately; it does not unwind DQ scopes. `Errno()` reads the C runtime's error value. Interpret it only according to the native operation that just failed, since successful calls need not clear it.

See Runtime and Program Execution for a complete argument-handling example and target differences.

4.2.2 Raw Memory

```
func MemAlloc(allocbytes : int, zeromem : bool = false) -> pointer
func MemFree(memptr : pointer)
func MemFreeRef(memptrvar : ref pointer)
func MemRealloc(memptr : pointer, newsize : uint) -> pointer
func MemReallocRef(memptr : ref pointer, newsize : uint)
func MemCopy(dest : pointer, src : pointer, size : uint)
func MemSet(dest : pointer, value : int, size : uint)
func MemBlockSize(memptr : pointer) -> uint
```

`MemAlloc` returns raw storage, optionally zeroed. Check for `nil` before use. `MemFreeRef` frees an allocation and clears the pointer variable. `MemFree` does not clear other aliases. Neither function runs object destructors.

```
use print

func *Main():
    var buffer : ^byte = ^byte(MemAlloc(64, true))
    if buffer == nil:
        exit_code = 1
        return
    endif
    buffer^ = 42
    PrintLn("First byte: {}", [buffer^])
    MemFree(buffer)
    buffer = nil
endfunc
```

`MemCopy` copies bytes and handles overlapping source and destination ranges. `MemSet` fills a block with the low byte of `value`. Sizes are measured in bytes; these operations do not check the size or lifetime of the supplied storage. Do not use byte copying to duplicate managed strings, arrays, or owning fields.

`MemRealloc` can move an allocation. On allocation failure with a nonzero size, it returns `nil` while the old block remains allocated; keep the old pointer until the result has been checked. `MemReallocRef` directly replaces the pointer and therefore does not preserve that recovery handle on failure.

`MemBlockSize` reports the allocator's usable block size on Linux and Windows; the fallback on other targets returns zero. It is not a portable way to track application buffer lengths.

Prefer managed strings, arrays, and objects for ordinary storage. See Memory Management and Autofree for cleanup and owning pointers.

4.2.3 Exceptions and Backtraces

With exceptions enabled, `Exception` supplies a formatted message and a captured backtrace. A handler can inspect `error.message` and call `error.PrintBacktrace()`. Compile with `-g -O0` for useful source-line diagnostics. The complete rules and examples are in Exceptions and Error Handling.

4.3. `print`

Import `print` to write text to standard output:

```
use print

func *Main():
    Print("Items: ")
    PrintLn("{}", [3])
    PrintLn()
endfunc
```

This writes `Items: 3`, followed by an empty line.

4.3.1 Functions

```
func Print(fmt : strslice)
func Print(fmt : strslice, args : varargs)
func PrintLn()
func PrintLn(fmt : strslice)
func PrintLn(fmt : strslice, args : varargs)
func PrintFlush()
func PrintSetUnbuffered()
```

`Print` writes without a newline. `PrintLn` appends a newline; its no-argument form writes only that newline.

The one-argument forms write text literally. Braces have a formatting meaning only in the forms that take a `varargs` argument.

```
PrintLn("{name}")             // prints {name}
PrintLn("{} = {:04X}", ["id", 255]) // prints id = 00FF
```

Pass one value directly or group multiple values in a bracketed list. See Text Formatting for widths, precision, number formats, escaping braces, and formatting error behavior.

4.3.2 Buffering

`PrintFlush()` asks the C runtime to flush standard output. This is useful for a prompt without a newline. `PrintSetUnbuffered()` disables stdout buffering via the runtime; configure buffering before normal output begins.

These functions use the shared stdout sink and formatter. For custom output sinks or building a string instead of printing, see Text Formatting and String Functions.

4.4. `file`

The `file` module provides simple file, directory, and path helpers.

```
use file

FileWriteText("hello.txt", "hello\n")
var text : str = FileReadText("hello.txt")
```

Errors raise `EFileError`.

4.4.1 `OFile`

`OFile` wraps a libc file handle and closes it from its destructor.

```
var f <- OFile()
f.Open("data.bin", "rb")
var bytes : [*]byte = f.ReadBytes()
f.Close()
```

Method / property	Meaning
`Open(name, mode)`	open with a libc-style mode such as `"r"`, `"rb"`, `"w"`
`Close()`	close if open
`Opened()` / `.opened`	true when a handle is open
`Size()` / `.size`	file size, preserving current position
`Seek(pos)` / `.position = pos`	seek from file start
`CurPos()` / `.position`	current offset
`Read(ptr, size)`	read bytes into memory
`ReadText()`	read the whole file as `str`
`ReadBytes()`	read the whole file as `[*]byte`
`Write(ptr, size)`	write bytes from memory
`WriteText(text)`	write text
`WriteBytes(bytes)`	write a byte slice

4.4.2 Convenience Functions

```
FileRead(name, ptr, maxlen)
FileReadText(name)
FileReadBytes(name)

FileWrite(name, ptr, len)
FileWriteText(name, text)
FileWriteBytes(name, bytes)

FileExists(name)
FileRemove(name)
```

`FileExists` returns false for directories. `FileRemove` removes an existing file and raises `EFileError` for an empty, missing, or unremovable file.

4.4.3 Directories

```
DirExists("out")
DirCreate("out/nested", true)
DirRemove("out/nested")
```

`DirCreate(path, true)` creates missing parent directories. Without `true`, an existing directory is an error.

4.4.4 Path Helpers

```
PathSlashSeparators(path)
PathPlatformSeparators(path)
PathToAbsolute(path)
GetCurrentPath()
SetCurrentPath(path)
PathDirName(path)
PathFileName(path)
PathFileExt(path)
```

`PathSlashSeparators()` normalizes separators to `/`, while `PathPlatformSeparators()` uses `\\` on Windows and `/` elsewhere. `PathToAbsolute()` resolves a relative path against the current directory and normalizes `.` and `..` path segments. `GetCurrentPath()` returns the absolute current working directory. `SetCurrentPath(path)` changes the process working directory.

4.5. `json`

The `json` module provides a mutable JSON tree with parsing and serialization.

```
use json

var j <- OJson()
j.AddStr("name", "dq")
j.AddNum("version", 1)
PrintLn("{}", [j.pretty_json])
```

Errors raise `EJson`.

4.5.1 Node Kinds

```
enum NJsonNodeKind = (nkNull, nkBool, nkNumber, nkString, nkObject, nkArray)
```

An `OJson` node stores its kind, optional scalar value, parent pointer, name, and children.

4.5.2 Building Trees

```
var root <- OJson()
root.AddStr("title", "example")
root.AddBool("ok", true)

var items : OJson = root.AddArr("items")
items.AddStr("", "first")
items.AddNum("", 2)

var cfg : OJson = root.ForcePath("settings/theme")
cfg.as_str = "dark"
```

Method	Meaning
`AddNull(name)`	append null child
`AddBool(name, value)`	append boolean child
`AddStr(name, value)`	append string child
`AddNum(name, value)`	append number child
`AddObj(name)`	append object child
`AddArr(name)`	append array child
`Clear()`	delete all children

For array items, the name is usually `""`.

4.5.3 Reading And Writing Values

Property / method	Meaning
`.name`	node name, or `"0"` for root
`.parent`	parent node
`.as_str`	read/write scalar as string
`.as_num`	read/write scalar as float
`.as_bool`	read/write scalar as bool
`.is_null`	true for JSON null
`.json`	compact JSON string
`.pretty_json`	indented JSON string
`.length`	child count
`.child[index]`	child by index, or null

```
var title : str = root.ChildByName("title").as_str
```

4.5.4 Finding Nodes

```
var jv : OJson

// most practical:
if root.Find("settings.theme", jv):
    PrintLn("theme={}", [jv.as_str])
endif

// traditional:
jv = root.FindNode("settings/theme")
if jv <> nil:
    PrintLn("theme={}", [jv.as_str])
endif
```

Paths can use `.` or `/` between object names. `ForcePath` creates missing object nodes.

4.5.5 Parsing And Files

```
var j <- OJson()
j.Parse('{"answer":42}')
j.LoadFromFile("config.json")
j.SaveToFile("out.json", true)
```

The root JSON text must be an object or array. Strings support JSON escapes, including Unicode escapes and surrogate pairs.

4.6. `threads`

`threads` provides native threads, atomic values, spinlocks, thread notifications, and a bounded message channel for hosted Linux and x86_64 Windows programs.

```
use threads
```

All the types below are available through this import. The `.dqh` and `.dqi` files are parts of the `threads` module, not separate modules to import. On Linux, the package links `pthread` and `atomic`; their libraries must be available to the linker. `OThread.IsRunning()` uses `pthread_tryjoin_np`, and `OThreadEvent` and channel waits use futexes. On 32-bit Linux, waits require `futex_time64` (Linux 5.1 or later).

On Windows, the package uses `_beginthreadex`, interlocked atomic operations, and `WaitOnAddress` / `WakeByAddressSingle`. It links `kernel32` and `synchronization` from the llvm-mingw toolchain, without requiring `pthread` or `libatomic`. Address-based waits require Windows 8 / Windows Server 2012 or later. The same thread, channel, spinlock, and event APIs work on both platforms.

Type / function	Purpose
`OThread`	start and join a native thread
`OMsgChannel`	bounded FIFO shared by producers and consumers
`SChMsg`	fixed 16-byte channel message
`OAtomicUint32`, `OAtomicUint64`, `OAtomicPtr`	atomic value access
`OSpinLockVar`, `OSpinLockHolder`	shared lock and scope-based lock holder
`OThreadEvent`	notification used when waiting for a shared condition
`SchedYield()`	give other runnable threads an opportunity to run

4.6.1 Threads

`OThread` starts one thread at a time. Choose a free function, a bound object method, or the thread object's virtual `Run` method:

Method	Meaning
`StartFunc(afunc : FThreadFunc, aarg : pointer) -> bool`	run a free function
`StartMethod(amethod : FThreadMethod, aarg : pointer) -> bool`	run a bound object method
`StartSelf(aarg : pointer) -> bool`	run this object's `Run(aarg)`
`Run(aarg : pointer) -> pointer`	virtual entry point for `StartSelf`; default returns `nil`
`Join()`	wait for the thread to finish and collect its return value
`IsRunning() -> bool`	try to join without waiting; collect the return value if finished

The callback types are:

```
type FThreadFunc   = func(arg : pointer) -> pointer
type FThreadMethod = func(arg : pointer) -> pointer of object
```

Each start method returns `false` if the wrapper still has an unjoined thread or native thread creation fails. A successful start sets `thread_id` (a pthread ID on Linux, a numeric thread ID on Windows); the callback's full pointer result is available in `retval` after joining. `Join()` on an idle wrapper does nothing. After a successful join, the same wrapper can start another thread.

```
use threads
use print

func ThreadMain(arg : pointer) -> pointer:
```

```
    return arg
endfunc

func *Main():
    var value : int = 42
    var thread <- OThread()
    if not thread.StartFunc(ThreadMain, %value):
        Exit(1)
    endif
    thread.Join()
    PrintLn('result = {}', [^int(thread.retval)^])
endfunc
```

For a bound method, use `thread.StartMethod(worker.ThreadMain, arg)`. For a subclass of `OThread`, override `Run` and call `worker.StartSelf(arg)` as shown in the channel worker example.

`IsRunning()` has a side effect: it reaps a completed thread just as `Join()` does, including closing the native thread handle on Windows. On Linux it also returns `false` on try-join errors other than "busy"; on Windows a native wait error retains the running state. It is not an error-reporting API. `Join()` returns no status; if the native join fails, the wrapper retains its running state.

Threads share the process address space. The wrapper, bound-method receiver, and argument storage must remain alive until the thread finishes. `OThread` does not automatically stop or join from a destructor. Manage each wrapper from one controlling thread; its lifecycle fields are not synchronized. Join before destroying a worker or reading results that only that worker writes.

4.6.2 Message Channels

`OMsgChannel` is a ring-buffer FIFO with a fixed size chosen at construction. It supports multiple producers and multiple consumers with its default `locking = true` setting. Each successful `Get` removes one message for one consumer; it does not broadcast a message to all workers.

4.6.2.1 Capacity and Nonblocking Operations

```
use threads
use print

func *Main():
    var channel <- OMsgChannel(64) // capacity: 64 messages
    var message : SChMsg = {msg: 1, par.u32: 7, data.u64: 12345}
    if not channel.Put(message):
        Exit(1)
    endif

    var received : SChMsg = {}
    if channel.Get(received):
        PrintLn('message {}, job {}, value {}',
                [received.msg, received.par.u32, received.data.u64])
    endif
endfunc
```

The constructor argument is the **usable message capacity**: `OMsgChannel(n)` holds at most `n` messages. The channel allocates one additional ring slot internally to distinguish a full queue from an empty queue. `OMsgChannel(1)` holds one message; `OMsgChannel(0)` always rejects puts and gets. The argument is `uint32`; `0xFFFFFFFF` raises `Exception` because the additional slot would overflow the internal slot count. Allocation must also fit available memory.

Method	Success	Failure
`Put(amsg : SChMsg) -> bool`	copy the message into the queue; return `true`	full queue; return `false` without changing queued messages
`Get(rmsg : refout SChMsg) -> bool`	remove the oldest message into `rmsg`; return `true`	empty queue; return `false` and leave `rmsg` unchanged
`PutWaitMillis(amsg : SChMsg, atimeoutms : int) -> bool`	enqueue, waiting for space if needed	return `false` if waiting ends without space
`GetWaitMillis(rmsg : refout SChMsg, atimeoutms : int) -> bool`	dequeue, waiting for a message if needed	return `false` if waiting ends without a message; leave `rmsg` unchanged

`Put` and `Get` do not wait for queue space or data. They can still contend for the internal spinlock. FIFO describes the order of successful enqueues and dequeues; concurrent producers have no predetermined relative order, and workers can finish processing messages in a different order.

Set `locking` before sharing the channel. Keep it `true` for concurrent use. Disabling it removes synchronization of the buffer and indices: it requires single-threaded access or external synchronization of every buffer operation. The volatile indices alone do not establish a safe lock-free channel, even with one producer and one consumer.

4.6.2.2 Message Layout and Ownership

`SChMsg` has a message identifier, a 4-byte parameter union, and an 8-byte data union. Choose the union view that matches your own message protocol:

Field	Available views
`msg`	`uint32` identifier
`par`	`bytes : [4]byte`, `i16 : [2]int16`, `u16 : [2]uint16`, `i32 : int32`, `u32 : uint32`, `f32 : float32`
`data`	`bytes : [8]byte`, `i16 : [4]int16`, `u16 : [4]uint16`, `i32 : [2]int32`, `u32 : [2]uint32`, `f32 : [2]float32`, `i64 : int64`, `u64 : uint64`, `f64 : float64`, `ptr : pointer`

The views of each union overlap; writing `data.u64` replaces the same storage as `data.f64` or `data.ptr`. There is no automatic payload type tag. Initialize the structure, for example with `var message : SChMsg = {}`, and agree on which fields each `msg` identifier uses.

Messages are copied by value. A successful `Put` lets the sender reuse its local `SChMsg`, but `data.ptr` copies only an address. The channel neither copies the pointed-to data nor frees it. Keep payload storage alive until the consumer is done, and define who releases it. If enqueueing fails, no message or payload has been handed to a consumer. Destroying a channel also does not free payloads referenced by messages still in the queue.

4.6.2.3 Waiting and Timeouts

The timeout argument has the same meaning for both waiting methods:

`atimeoutms`	Behavior
`0`	one immediate `Put` or `Get` attempt
negative	wait indefinitely until the operation succeeds or the wait fails
positive	attempt immediately, then wait using one monotonic deadline

An available operation succeeds immediately even with a positive timeout. Failed attempts and signal interruptions reuse the original deadline; they do not restart the timeout. Very large timeouts saturate the deadline rather than overflowing it. Scheduling and internal lock contention can delay the return beyond the requested timeout, so it is not a strict bound on execution time.

Successful enqueues signal a waiting consumer; successful dequeues signal a waiting producer. Waiters sleep using process-private Linux futexes or Windows address-based waits. A sequence snapshot taken before checking the queue prevents a notification between that check and the kernel wait from being missed. A wake only causes another queue attempt; a different worker may already have taken the available message or space. Native wait errors other than an interruption or changed sequence also produce `false`, so the result does not distinguish a timeout from such an error.

4.6.2.4 Distributing Work

This complete example sends jobs to one worker and then sends an explicit stop message. The worker's result is read after joining:

```
use threads
use print

const MSG_WORK : uint32 = 1
```

```
const MSG_STOP : uint32 = 2

object OWorker(OThread):
    channel : OMsgChannel
    total : uint64 = 0

    func *Create(achannel : OMsgChannel):
        channel = achannel
    endfunc

    func Run(arg : pointer) -> pointer [[override]]:
        use *
        var job : SChMsg
        while channel.GetWaitMillis(job, -1):
            if job.msg == MSG_STOP:
                break
            endif
            if job.msg == MSG_WORK:
                total += job.data.u64
            endif
        endwhile
        return nil
    endfunc
endobject

func *Main():
    var channel <- OMsgChannel(4) // capacity: four messages
    var worker <- OWorker(channel)
    if not worker.StartSelf(nil):
        Exit(1)
    endif
    var job : SChMsg = {msg: MSG_WORK, par.u32: 0, data.u64: 0}
    for i : int = 1 to 10:
        job.data.u64 = i
        if not channel.PutWaitMillis(job, 1000):
            Exit(1)
        endif
    endfor
    job.msg = MSG_STOP
    if not channel.PutWaitMillis(job, 1000):
        Exit(1)
    endif
    worker.Join()
    PrintLn('total = {}', [worker.total]) // 55
endfunc
```

Waiting on a full queue gives the producer backpressure. To use several workers, give each the same channel and enqueue one stop message per worker after all producers have finished enqueueing jobs. FIFO places those stops after the jobs; join every worker before inspecting its results or releasing the channel. Processing completion order and distribution among workers are not guaranteed.

There is no channel close, cancellation, or broadcast API. An indefinite wait needs a cooperating producer or consumer to make progress. Use finite waits if workers need to check a separate shutdown condition. Do not destroy a channel while operations or waits are still running. Channels and events operate among threads in one process, not between processes.

4.6.3 Atomic Values

Construct an atomic wrapper with its initial value, for example `var flag <- OAtomicUint32(0)` or `var current <- OAtomicPtr(nil)`. All three types expose `Load()`, `Store(value)`, and `Exchange(value)`. `Exchange` replaces the value and returns its previous value. The integer wrappers also expose `Add(value)`, which returns the value **before** addition; unsigned overflow wraps.

`Load()`, `Store(value)`, `Exchange(value)`, and `Add(value)` are sequentially consistent and take no memory-order argument. Loads acquire published data, stores release earlier writes, and exchanges and additions do both. These operations also participate in a single ordering across atomic variables.

All three types provide `LoadRelaxed()`. The integer types also provide `AddRelaxed(value)`, which returns the value before addition and wraps on unsigned overflow, just like `Add(value)`. For example, use `counter.AddRelaxed(1)` and `counter.LoadRelaxed()` for independent statistics. Relaxed operations remain atomic but do not synchronize other shared storage; do not use a relaxed load of a ready flag to acquire data published by another thread.

On Windows, all these operations use interlocked functions with full memory barriers, providing stronger ordering for the relaxed methods. Linux uses the method's specified ordering. Do not rely on the stronger Windows behavior in portable code.

Each method call is atomic; a sequence of calls is not a single atomic operation. Use a lock when several accesses must happen together. Atomic pointer publication still requires separate management of the pointed-to allocation's lifetime.

`OAtomicUint32.Address()` exposes the value's address for native address-based waits. Keep the wrapper alive while using this address, and access its value through the atomic methods.

`SchedYield() -> int32` wraps `sched_yield` on Linux and `Sleep(0)` on Windows; it returns zero on Windows. Yielding does not synchronize shared data.

4.6.4 Spinlocks

`OSpinLockVar` inherits the atomic integer constructor. Initialize an unlocked lock with `OSpinLockVar(0)`. `Lock()` acquires it, yielding while contended; `Unlock()` releases it. The lock is not recursive and has no timeout or fairness guarantee. Keep protected sections short.

`OSpinLockHolder(lock)` acquires in its constructor and releases in its destructor. An embedded local holder provides scope-based release, including early return and exception unwinding:

```
use threads

var lock <- OSpinLockVar(0)
var counter : int = 0

func IncrementCounter():
    var holder <- OSpinLockHolder(lock)
    ++counter
endfunc
```

Every concurrent access to the protected data must use the same lock. The lock must outlive all holders and users. A holder does not own the lock; explicitly delete a heap-allocated holder to release it, or use an embedded local as above.

4.6.5 Thread Events

`OThreadEvent` is a notification primitive used by the channel. It does not store a job or represent a persistent boolean condition.

Method	Meaning
`Snapshot() -> uint32`	read the notification sequence
`Signal()`	advance the sequence and wake at most one registered waiter
`WaitUntilNanos(asequence : uint32, adeadline : int64) -> bool`	sleep only while the sequence still matches; negative deadline means indefinite wait

For a finite deadline, pass an absolute monotonic time such as `NanoTime() + 100000000` for 100 ms, with `use time`. `true` means recheck the shared condition: the thread woke, the sequence changed, or a signal interrupted the wait. `false` means the deadline expired or another native wait error occurred. An already-expired deadline returns `false` before checking for a changed sequence.

Use the event with a separately synchronized condition:

1. Take a sequence snapshot.
2. Check the condition under its lock or with the appropriate atomic operations.
3. If satisfied, continue with the work.
4. Otherwise, wait using that snapshot and the original absolute deadline.
5. After a `true` result, repeat from the snapshot step.

The thread changing the condition updates it first, then calls `Signal()`. Taking the snapshot before checking ensures a notification in between checking and sleeping is observed. A signal before the snapshot does not count as a pending notification; the shared condition is the source of truth. Keep the event alive until every waiter and signaler has finished.

4.6.6 Examples and Checks

The source examples are in `stdpkg/threads/examples/`. Run these commands from the repository root:

Example	Shows
`thread_test.dq`	free-function threads, a `Run` override, completion polling; also includes a bound-method example
`wait_test.dq`	immediate, finite, indefinite, and very large waits; unchanged output on failure; notification races and signal interruptions
`stress_test.dq`	full/empty queues, FIFO and ring wraparound, one/many producers and consumers, corruption/duplicate/missing-message checks
`work_distribution.dq`	a bounded queue feeding a worker pool, stop messages, and deterministic Monte Carlo jobs compared against one worker

```
build/dq-run stdpkg/threads/examples/thread_test.dq
build/dq-run stdpkg/threads/examples/wait_test.dq
build/dq-run stdpkg/threads/examples/stress_test.dq -- 4 50000 256 15
build/dq-run stdpkg/threads/examples/stress_test.dq -- 4 20000 1 15 100
build/dq-run -O3 --lto stdpkg/threads/examples/work_distribution.dq -- 4 256 1000000
```

The stress tester's arguments are worker count, messages per producer, queue capacity, scenario timeout in seconds, and operation wait timeout in milliseconds. Defaults are the detected online processor count, 50,000 messages, capacity 256, 15 seconds, and zero (nonblocking operations). Its consumer bookkeeping uses roughly `producers * consumers * messages_per_producer` bytes. Exit status is 0 for success, 1 for failed checks or timeout, and 2 for invalid arguments or thread creation failure.

The work-distribution arguments are maximum workers, job count, and samples per job; defaults are the detected processor count, 256, and 1,000,000. It compares one worker with progressively larger pools, using the same work each time. Timing includes creation, dispatch, computation, and joining, with validation afterward. Use `-O3 --lto` for speed comparisons.

On a Linux host with llvm-mingw and Wine installed, `make test-windows` cross-compiles and runs thread lifecycle, atomic, event, channel capacity, wait, stress, and work-distribution checks at `-O0` and `-O3 --lto=full`. See Windows application cross-compilation.

4.7. `time`

`time` provides wall-clock timestamps, monotonic clocks, date/time formatting, and sleeps for hosted Linux and Windows programs.

```
use time
```

4.7.1 Choosing a Clock

Function	Return type	Unit	Use
`Time()`	`float64`	seconds since the Unix epoch (1970-01-01 00:00:00 UTC)	dates, timestamps, and wall-clock deadlines
`MilliTime()`	`int64`	milliseconds	elapsed time and monotonic deadlines
`MicroTime()`	`int64`	microseconds	elapsed time and monotonic deadlines
`NanoTime()`	`int64`	nanoseconds	elapsed time and monotonic deadlines

`Time()` includes a fractional second. It reads the system wall clock, so clock corrections can change its value forwards or backwards. Use it when a value must correspond to a calendar date, such as a log timestamp.

The monotonic clocks have a system-defined origin unrelated to the Unix epoch. Subtract two readings from the same clock to measure a duration, or add a duration to form a deadline. Their values are unaffected by wall-clock changes; do not pass them to the date/time formatters or store them as calendar timestamps. The unit names describe the returned scale, not a guarantee of clock resolution.

4.7.1.1 Measuring Elapsed Time

```
use print
use time

func *Main():
    var started : int64 = MicroTime()
    SleepMillis(20) // Replace with the operation to measure.
    var elapsed : int64 = MicroTime() - started
    PrintLn('Elapsed: {} us ({:.3f} ms)', [elapsed, elapsed / 1000])
endfunc
```

The elapsed value can exceed the requested sleep because the operating system decides when the thread runs again.

4.7.2 Formatting Dates and Times

All three formatters take the format first and a `Time()`-style timestamp second:

```
func FormatTime1(fmt : rostr, atime : float64) -> str ..
func FormatTime2(fmt : rostr, atime : float64) -> str ..
func FormatTimeUtc(fmt : rostr, atime : float64) -> str ..
```

Function	Time zone	Format language
`FormatTime1`	system local time	Delphi-style tokens
`FormatTime2`	system local time	native `strftime` directives, plus `%f` for microseconds
`FormatTimeUtc`	UTC	native `strftime` directives; `%f` is not expanded

The timestamp itself has no local-time offset. `FormatTime1` and `FormatTime2` apply the system's local time-zone and daylight-saving rules during conversion. `FormatTimeUtc` formats the same instant in UTC. None of the formatters accepts an explicit time-zone argument.

```
use print
use time

func *Main():
    var now : float64 = Time()
    PrintLn('Local: {}', [FormatTime1('yyyy-mm-dd hh:nn:ss.zzz', now)])
    PrintLn('Local with microseconds: {}',
            [FormatTime2('%Y-%m-%d %H:%M:%S.%f', now)])
    PrintLn('UTC: {}', [FormatTimeUtc('%Y-%m-%dT%H:%M:%SZ', now)])

    // Fixed timestamp: 2024-01-01 00:00:00 UTC.
    PrintLn(FormatTimeUtc('%Y-%m-%d %H:%M:%S', 1704067200))
endfunc
```

The last line prints `2024-01-01 00:00:00` regardless of the local time zone. Only append the literal `Z` when formatting UTC.

4.7.2.1 Delphi-Style Tokens (`FormatTime1`)

Tokens are case-insensitive. Prefer `nn` for minutes to distinguish them from the month token `mm`.

Token	Meaning
`yy`, `yyyy`	two-digit year, full year
`m`, `mm`	month without padding, two-digit month
`mmm`, `mmmm`	abbreviated month name, full month name
`d`, `dd`	day of month without padding, two-digit day
`ddd`, `dddd`	abbreviated weekday name, full weekday name
`ddddd`	locale date representation
`h`, `hh`	hour without padding, two-digit hour; normally 24-hour time
`n`, `nn`	minute without padding, two-digit minute
`s`, `ss`	second without padding, two-digit second
`z`, `zz`, `zzz`	milliseconds padded to at least one, two, or three digits
`am/pm`, `ampm`	locale AM/PM marker; switches hours to 12-hour time
`a/p`	`a` or `p` (`A` or `P` for uppercase token); switches hours to 12-hour time
`c`	locale date and time representation
`t`, `tt`	locale time representation

`m` and `mm` mean minutes immediately after an hour token, optionally separated by a colon: `hh:mm:ss` works, but `hh:nn:ss` is clearer. Text inside matching single or double quotes is copied literally; other characters are copied unless they form a recognized token. For example:

```
PrintLn(FormatTime1('yyyy-mm-dd "at" hh:nn:ss', Time()))
PrintLn(FormatTime1('hh:nn am/pm', Time()))
```

This is the token set implemented by DQ, rather than the entire Delphi formatting API. Name and locale tokens use the native C library. Unpadded numeric tokens use native `strftime` `%-` directives, whose support varies by platform; prefer the padded forms for portable output.

4.7.2.2 `strftime` Directives (`FormatTime2` and `FormatTimeUtc`)

Directive	Meaning
`%Y`, `%y`	full year, two-digit year
`%m`, `%d`	two-digit month, two-digit day of month
`%H`, `%I`	two-digit 24-hour hour, two-digit 12-hour hour
`%M`, `%S`	two-digit minute, two-digit second
`%a`, `%A`	abbreviated weekday name, full weekday name
`%b`, `%B`	abbreviated month name, full month name
`%p`	locale AM/PM marker
`%j`	day of year, padded to three digits
`%w`	weekday number, Sunday = 0
`%x`, `%X`, `%c`	locale date, time, date and time
`%%`	literal percent sign
`%f`	six-digit microseconds; `FormatTime2` **only**

Other directives and flags are passed to the native C library, so their support and output can differ between platforms. In particular, `FormatTime2` does not implement all of Python's date/time API; it adds `%f` to native `strftime`.

Calendar conversion rounds the fractional second to the nearest microsecond, carrying into the next second when necessary. `FormatTime1` then takes the millisecond part for `z` tokens. `FormatTimeUtc` outputs no fractional seconds with the directives above.

An empty format returns an empty string. Conversion failures or a native formatting result that cannot be produced within the internal buffer limit also return an empty string; the functions do not return a separate error status.

4.7.3 Sleeping for a Duration

Function	Argument type	Unit
`SleepSecs(asecs)`	`float64`	seconds; fractions are accepted
`SleepMillis(ams)`	`int64`	milliseconds
`SleepMicros(aus)`	`int64`	microseconds
`SleepNanos(ansecs)`	`int64`	nanoseconds

These functions block the calling thread and return no value. Zero or negative durations return immediately. `SleepSecs` rounds its duration to nanoseconds. On Windows, positive durations are rounded up to whole milliseconds before calling the native sleep API. On Linux, the package uses `nanosleep` and retries after signal interruptions with the remaining duration.

```
SleepSecs(0.05)   // 50 milliseconds.
SleepMillis(50)
SleepMicros(500)
SleepNanos(500000)
```

Actual wake-up time depends on OS timer resolution and scheduling. A nanosecond argument does not imply nanosecond scheduling accuracy. Keep durations within the `int64` range after conversion to nanoseconds; conversion does not saturate on overflow.

4.7.4 Sleeping Until a Deadline

Function	Deadline type	Matching clock
`SleepToTime(asecs)`	`float64`	`Time()`
`SleepToMilliTime(ams)`	`int64`	`MilliTime()`
`SleepToMicroTime(aus)`	`int64`	`MicroTime()`
`SleepToNanoTime(ansecs)`	`int64`	`NanoTime()`

These functions take an **absolute deadline**, not a duration, and return no value. Use a reading from the matching clock plus the desired delay:

```
SleepToTime(Time() + 0.05)
SleepToMilliTime(MilliTime() + 50)
SleepToMicroTime(MicroTime() + 50000)
SleepToNanoTime(NanoTime() + 50000000)
```

A deadline already reached, or a nonpositive deadline, returns immediately. Prefer a monotonic clock for timeouts and periodic work. `SleepToTime` follows wall-clock time, so clock adjustments can affect the wait. Sleep functions do not expose native errors to the caller; they are not a hard real-time guarantee.

4.7.4.1 Periodic Work Without Accumulating Work-Time Drift

Advance the previous deadline by a fixed interval. Sleeping for an interval after every iteration would add each iteration's work time to the period.

```
use print
use time

func *Main():
    const period_ms : int64 = 100
    var deadline : int64 = MilliTime()
    for tick : int = 0 count 5:
        deadline += period_ms
        SleepToMilliTime(deadline)
        PrintLn('Tick {} at {}',
                [tick + 1, FormatTime1('hh:nn:ss.zzz', Time())])
    endfor
endfunc
```

The first tick is scheduled one period after the initial clock reading. If work takes longer than the period, past deadlines return immediately and this loop catches up. Applications that should skip missed ticks need an explicit policy for advancing the deadline past the current time.

4.7.5 Running the Examples

The complete programs above are also provided in `stdpkg/time/examples/`. Run them from the repository root:

```
build/dq-run stdpkg/time/examples/elapsed.dq
build/dq-run stdpkg/time/examples/formatting.dq
build/dq-run stdpkg/time/examples/periodic.dq
```

With an installed release, use `dq-run` in place of `build/dq-run`.

4.8. `nanonet`

`nanonet` is a small event-driven networking package. It currently targets the hosted socket runtime used by the project examples.

```
use nanonet/nano_sockets
use nanonet/nano_http
```

4.8.1 Sockets

`nano_sockets` provides non-blocking TCP/UDP sockets, an event watcher, and a server/connection framework.

Type	Purpose
`ONanoSocket`	low-level socket wrapper
`OSocketWatcher`	event loop around platform socket events
`OSConnection`	base server-side connection object
`ONanoServer`	listener and connection manager
`ENanoNet`	socket/server exception

Server connections are object types derived from `OSConnection`. Override `HandleInput` and optionally `HandleOutput`.

```
object OMyConnection(OSConnection):
    func HandleInput(aobj : Object) [[override]]
endobject

func OMyConnection.HandleInput(aobj : Object) [[override]]:
    var buf : [256]byte
    var n : int = sock.Recv(%buf[0], SizeOf(buf))
    if n <= 0:
        Close()
    endif
endfunc
```

Create a server with the connection type and listen port:

```
var svr <- ONanoServer(OMyConnection, 8080)
svr.InitListener()
while true:
    svr.WaitForEvents(1000)
endwhile
```

4.8.2 HTTP

`nano_http` builds an HTTP request/response layer on top of `nano_sockets`.

Type	Purpose
`OStrMap`	simple string map used for headers, query values, cookies
`OSConnectionHttp`	HTTP connection base class
`ONanoHttpServer`	HTTP server with content-type map

Override `ProcessRequest()` to handle dynamic responses:

```
object OConn(OSConnectionHttp):
    func ProcessRequest() -> bool [[override]]
endobject

func OConn.ProcessRequest() -> bool [[override]]:
    response_headers["Content-Type"] = "text/plain"
    response = "hello"
```

```
    return true
endfunc
```

Important request fields on `OSConnectionHttp` include:

Field	Meaning
`method`	request method, uppercase
`uri`	path without query string
`url`	original request URL
`getstr`	raw query string
`http_ver`	HTTP version text
`keep_alive`	connection reuse flag
`ucheaders`	request headers with uppercase keys
`qsvars`	parsed query string values
`cookies`	parsed cookie values

Response fields:

Field	Meaning
`response_code`	status code, default `200`
`response_headers`	output headers
`response`	in-memory response body
`full_content_length`	length for streamed file responses

4.8.3 Static Files

`HandleStaticFiles(root)` serves files below a directory and rejects unsafe `..` paths.

```
func OConn.ProcessRequest() -> bool [[override]]:
    if HandleStaticFiles("public"):
        return true
    endif
    response_code = 404
    response = "not found"
    return true
endfunc
```

The HTTP server maps common extensions such as `.html`, `.css`, `.js`, `.png`, `.jpg`, `.svg`, and `.ico` to content types. Unknown extensions use `application/octet-stream`.

See the examples in `examples/nanonet/`, especially `http_static_files.dq`.

4.9. `shell`

`shell` runs child processes, captures their output, supplies standard input, and reads or updates environment variables.

```
use shell
```

The package supports hosted Linux and x86_64 Windows. Linux uses `fork`, `execvp`, and `poll`; Windows uses Unicode `CreateProcessW`, redirected pipes, and job objects. Both use `threads` for background execution. Windows stdin is fed by a separate writer thread so large input cannot block output capture.

4.9.1 Running Commands

Create an `OShellRunner` and use `Run` for a foreground command:

```
use print
use shell

func *Main():
    var runner <- OShellRunner()
#ifdef WINDOWS
    var code : int = runner.Run('echo hello& echo warning >&2& exit /b 7', false)
#else
    var code : int = runner.Run('printf hello; printf warning >&2; exit 7', false)
#endif
    PrintLn('exit: {}', [code])
    PrintLn('stdout: {}', [runner.out])
    PrintLn('stderr: {}', [runner.err])

    // Argument arrays launch an executable directly.
#ifdef WINDOWS
    runner.RunWithInput(['sort'], 'pear\napple\n')
#else
    runner.Run(['printf', '%s', 'literal $HOME; *.dq'])
#endif
    PrintLn(runner.out)

    try:
#ifdef WINDOWS
        runner.Run('exit /b 3')
#else
        runner.Run('exit 3')
#endif
    except EShellResult as e:
        PrintLn('Caught exit {}: {}', [e.code, e.message])
    endtry
endfunc
```

The first command returns `7`, with `hello` in `out` and `warning` in `err` (including the newlines written by Windows `echo`). The direct-execution example sorts input on Windows and prints a literal argument on Linux. The last command demonstrates the default exception behavior for a nonzero exit code.

4.9.1.1 Command Strings and Argument Arrays

Each command-starting method has two overloads:

Argument	Execution
`acmd : rostr`	run through the platform's command interpreter; shell quoting, expansion, pipes, and redirection apply
`args : []rostr`	execute `args[0]` with the remaining elements as individual arguments

On Linux, command strings run through `/bin/sh -c`. On Windows, they use `COMSPEC` (falling back to `cmd.exe`) with `/d /s /c`: AutoRun commands are disabled, and the command's own quoting is preserved. Shell syntax differs: use `;` and `$NAME` on Linux, versus `&` and `%NAME%` in Windows command strings. Windows built-ins such as `dir`, and `.bat` or `.cmd` scripts, need a command string. PowerShell is not selected automatically.

For argument arrays, Linux searches `PATH` for names without a slash. Windows uses the native `CreateProcessW` executable search and `.exe` suffix rules. An explicit executable path avoids the search. The child inherits the caller's working directory and environment; there is no per-command directory or environment option.

Use argument arrays for values that should be passed literally. A filename with spaces is one array element and needs no additional quoting. In an argument array, `|`, `>`, `*`, and `$HOME` are ordinary arguments unless the executable itself interprets them. Windows arguments are encoded using the usual C runtime quoting rules: empty values, spaces, embedded quotes, and trailing backslashes are preserved. Programs with their own command-line parsers can differ; use command strings for `cmd.exe` syntax. Executable names, arguments, and environment names/values are converted between DQ's UTF-8 strings and Windows UTF-16 APIs.

4.9.2 Foreground API

Method	Behavior
`Run(acmd : rostr, exitcerr : bool = true) -> int`	execute a shell command and wait
`Run(args : []rostr, exitcerr : bool = true) -> int`	execute an argument array and wait
`RunWithInput(acmd : rostr, ainput : strslice, exitcerr : bool = true) -> int`	execute a shell command with supplied stdin and wait
`RunWithInput(args : []rostr, ainput : strslice, exitcerr : bool = true) -> int`	execute an argument array with supplied stdin and wait

These calls collect output while the child runs and return its exit code after monitoring finishes. By default, a nonzero result raises `EShellResult` instead of returning normally. Pass `false` as `exitcerr` when nonzero exit codes are expected and inspect the returned value yourself. This only suppresses exit-code exceptions; process-start errors can still raise `EShellStart`. Calling a runner while it is busy returns `-1` without replacing the running command's state.

4.9.2.1 Captured Results

Field	Type	Meaning after completion
`out`	`str`	captured standard output
`err`	`str`	captured standard error
`output`	`str`	both streams merged in the monitor's pipe-read order
`exitcode`	`int`	process exit status; `-1` for Linux signal termination, Windows `Kill()`, or an unavailable result
`finished`	`bool`	initially `true`; set to `false` for a new run and `true` when monitoring finishes

A new run clears all three output strings and sets `exitcode` to `-1`. Output is captured as bytes without encoding or newline conversion. A Windows child can produce CRLF or a different text encoding from a Linux child; the runner does not normalize or decode it. Nothing is printed automatically: use `Print` or `PrintLn` to display the captured strings. The fields remain available when a foreground call raises `EShellResult`.

`output` does not guarantee the original write order across stdout and stderr. The streams use separate pipes, and reads can combine multiple writes or split them into chunks. The package does not provide a streaming callback or an output-size limit; it keeps the complete captures in memory. Since the child writes to pipes, its buffering can differ from an interactive terminal.

4.9.3 Supplying Standard Input

`Run` and `StartBg` leave stdin inherited from the caller. A command waiting for input can therefore block. Their `WithInput` variants supply a pipe instead: the input is copied into runner-owned storage, sent while output is drained, and the pipe is closed after all input has been written. An empty input string provides EOF without supplying any bytes.

```
use print
use shell

func *Main():
    var runner <- OShellRunner()
    runner.RunWithInput(['sort'], 'pear\napple\n')
```

```
    PrintLn('Sorted input:\n{}', [runner.out])

    runner.RunWithInput(['sort'], '')
    PrintLn('Empty input produced {} bytes', [runner.out.length])
endfunc
```

The same input mechanism is available through `StartBgWithInput` for background commands. The package offers one input string per run, rather than an interactive stdin-writing API. Supply input to commands that consume it; the API does not report how many input bytes the child actually accepted.

4.9.4 Background Commands

Method	Behavior
`StartBg(acmd : rostr) -> int`	start a shell command and a monitor thread
`StartBg(args : []rostr) -> int`	start an executable and a monitor thread
`StartBgWithInput(acmd : rostr, ainput : strslice) -> int`	start a shell command with supplied stdin
`StartBgWithInput(args : []rostr, ainput : strslice) -> int`	start an executable with supplied stdin
`Finished() -> bool`	poll completion and try to reap the monitor thread
`WaitFinishMs(ams : int) -> bool`	poll for completion for up to the requested milliseconds
`Kill()`	terminate the process group (Linux) or job (Windows), then wait up to 2000 ms for completion
`Join()`	inherited from `OThread`; wait for and reap the monitor thread without a timeout

The start methods return `0` when the process and monitor thread were started, or `-1` if the runner is busy or the monitor thread cannot be started. They can also raise `EShellStart`. A successful start returns neither a PID nor the child's exit code. Background completion does not raise `EShellResult`; inspect `exitcode` after waiting.

```
use print
use shell

func *Main():
    var runner <- OShellRunner()
#ifdef WINDOWS
    var command : rostr = 'echo finished'
#else
    var command : rostr = 'sleep 0.05; printf finished'
#endif
    if runner.StartBg(command) <> 0:
        PrintLn('Could not start background command')
        Exit(1)
    endif

    // Other work can run here while the command executes.
    if not runner.WaitFinishMs(1000):
        PrintLn('Timed out; stopping the command')
        runner.Kill()
    endif
    runner.Join()
    PrintLn('exit: {}, stdout: {}', [runner.exitcode, runner.out])

    if runner.StartBgWithInput(['sort'], 'pear\napple\n') <> 0:
        Exit(1)
    endif
    runner.Join()
    PrintLn(runner.out)
endfunc
```

`WaitFinishMs` uses a monotonic deadline and sleeps in 10 ms intervals. It returns `true` when completion is observed and `false` when the deadline is reached. Zero or negative values make an immediate completion check; a negative value does **not** mean wait indefinitely. Scheduling and polling can delay the return beyond the requested duration. A timeout leaves the command running.

4.9.4.1 Lifetime and Cleanup

Use one runner for one command at a time. Wait for and join a background run before reading its output, reusing the runner for a foreground call, or destroying the runner. Treat the public result fields as read-only, and do not inspect or change the strings while the monitor thread is writing them. Manage the runner from one controlling thread.

`OShellRunner` has no destructor that automatically kills or joins a running command. `Join()` is the simplest way to wait indefinitely; for a bounded wait, use `WaitFinishMs`, handle a timeout explicitly, and join before releasing the object.

`Kill()` is forceful termination, with no graceful shutdown phase. On Linux it sends `SIGKILL` to the process group created for the child; a descendant that leaves that group can escape the kill. On Windows, the child is created suspended and assigned to a job before it runs, so `Kill()` terminates the job's process tree, including descendants. Windows termination uses status `0xFFFFFFFF`, which the runner maps to `-1`; other Windows exit statuses are returned as their unsigned 32-bit values.

`Kill()` returns no success status, and its internal wait is bounded; use `Join()` to finish cleanup. Descendants keeping captured pipe ends open can postpone completion after the original child exits, because monitoring drains the pipes before collecting the exit result. Windows process and job handles are retained until reuse or destruction to avoid racing termination against handle cleanup.

4.9.5 Errors

Exception	When it is used
`EShellStart`	empty command or executable, or failure to set up pipes or create the process; Windows job/handle setup failures also use this exception
`EShellResult`	foreground command finishes with a nonzero result and `exitcerr` is `true`

Both derive from `Exception` and expose its `message` field. `EShellResult` also has `code : int`, matching the runner's exit result. Its constructors accept either `(acode : int, amsg : rostr)` or `(acode : int, afmt : strslice, avalues : varargs)`.

On Linux, failure to execute a program after the fork is reported by the child: it writes a diagnostic to captured stderr and exits with `127`, producing `EShellResult` by default in foreground execution. Windows `CreateProcessW` failure instead raises `EShellStart`, including the native error number. If the interpreter starts but cannot find a command, its own exit code and diagnostic are captured on either platform. A failed process launch leaves the runner available for reuse. The API does not separately report all native read, write, or wait failures.

4.9.6 Environment Variables

```
func GetEnv(name : rostr, default_value : rostr) -> str ..
func SetEnv(name : rostr, value : rostr) ..
```

`GetEnv` returns a copy of the variable's value, or `default_value` if the variable is absent. An existing empty value remains empty. `SetEnv` creates or overwrites a variable in the current process and raises `Exception` if the native update fails. There is no unset function in this package. Windows environment names are case-insensitive; Linux names are case-sensitive.

```
use print
use shell

func *Main():
    PrintLn('HOME: {}', [GetEnv('HOME', '(not set)')])
    SetEnv('DQ_SHELL_EXAMPLE', 'hello from DQ')
    PrintLn('Current process: {}', [GetEnv('DQ_SHELL_EXAMPLE', '')])

    var runner <- OShellRunner()
#ifdef WINDOWS
    runner.Run('echo %DQ_SHELL_EXAMPLE%')
#else
    runner.Run('printf "%s" "$DQ_SHELL_EXAMPLE"')
#endif
    PrintLn('Child process: {}', [runner.out])
endfunc
```

Environment changes apply to the current process and children started afterward; they do not change the parent shell or an already-running child. Since the environment is process-wide, perform setup before launching worker threads or coordinate concurrent access yourself.

4.9.7 Running the Examples

The complete programs above are available in `stdpkg/shell/examples/`. From the repository root:

```
build/dq-run stdpkg/shell/examples/capture.dq
build/dq-run stdpkg/shell/examples/input.dq
build/dq-run stdpkg/shell/examples/background.dq
build/dq-run stdpkg/shell/examples/environment.dq
```

With an installed release, use `dq-run` in place of `build/dq-run`. On Windows, the built tool path is `build\dq-run.exe`. The examples select platform-specific shell commands with `#ifdef WINDOWS` and use the system's `sort` executable for stdin demonstrations. Linux command strings also use `printf` and `sleep`.

4.10. Additional Standard Packages

The following modules are also shipped in `stdpkg`. They have focused source interfaces and examples rather than separate API manuals. Check each module's public declarations for exact signatures and supported platforms.

For timestamps, date/time formatting, and sleeps, see the dedicated `time` package manual. For child processes and environment variables, see the `shell` package manual.

Import	Purpose	Source or example
`math`	Mathematical functions backed by the native math library.	`math.dq`
`strparse`	Cursor-based text parsing.	`test_strparse.dq`
`collections/strstrmap`	String-to-string map.	`strstrmap.dq`
`db`	Shared database abstractions.	`db.dq`
`sqlite3`	SQLite binding using the SQLite library.	`sqlite_test.dq`
`mysql`	MySQL client binding.	`mysql_test.dq`
`socket`	Low-level socket declarations.	`socket.dq`
`libc/stdio`	C standard I/O declarations and stream helpers.	`stdio.dq`

4.10.1 Native Dependencies

A DQ import supplies declarations and code; it does not install a native library. Database applications need the matching client development libraries at link time and the corresponding shared libraries at runtime. For cross targets, the libraries must match the target ABI and be available in the selected sysroot or linker search paths.

Use C Interoperability when adding bindings, and Project Files for explicit linker inputs. For networking applications, start with the higher-level `nanonet` package.

5. Appendices

5.1. Example Programs

The repository contains small focused examples and larger demonstrations.

5.1.1 Language And Basics

Example	Shows
`examples/langdemo.dq`	broad language demo
`examples/langdemo_braces.dq`	brace-style syntax variant
`examples/basic/new_delete.dq`	manual allocation helpers
`examples/objects/dq_objects.dq`	objects, constructors, methods
`examples/objects/inheritance.dq`	object inheritance

5.1.2 Strings And Formatting

Example	Shows
`examples/string/embstrs.dq`	Embedded Strings
`examples/string/txtfmt.dq`	`Format`, `AddFmt`, stdout formatting

5.1.3 Files

Example	Shows
`stdpkg/file/examples/fileiotest.dq`	file IO, directory helpers

5.1.4 Time

See the `time` package documentation for clock selection, formatting tokens, and sleep behavior.

Example	Shows
`stdpkg/time/examples/elapsed.dq`	measuring a duration with a monotonic clock
`stdpkg/time/examples/formatting.dq`	local and UTC timestamps, milliseconds and microseconds
`stdpkg/time/examples/periodic.dq`	periodic work using absolute deadlines

5.1.5 Shell Commands

See the `shell` package documentation for process execution, output capture, and background command lifetimes. These examples support Linux and Windows.

Example	Shows
`stdpkg/shell/examples/capture.dq`	stdout/stderr capture, argument arrays, and exit-code handling
`stdpkg/shell/examples/input.dq`	supplying stdin and closing it with EOF
`stdpkg/shell/examples/background.dq`	background execution, timeouts, and cleanup

Example	Shows
`stdpkg/shell/examples/environment.dq`	reading, setting, and inheriting environment variables

5.1.6 Networking

Example	Shows
`stdpkg/nanonet/examples/simplebin_server.dq`	simple socket server
`stdpkg/nanonet/examples/http_static_files.dq`	HTTP server, query parsing, static files

5.1.7 Threads

See the `threads` package documentation for the APIs and example commands.

Example	Shows
`stdpkg/threads/examples/thread_test.dq`	starting threads and polling completion
`stdpkg/threads/examples/wait_test.dq`	message-channel waits and timeout checks
`stdpkg/threads/examples/stress_test.dq`	channel correctness with multiple producers and consumers
`stdpkg/threads/examples/work_distribution.dq`	distributing CPU work through a bounded channel

Run an example with:

```
dq-run examples/string/txtfmt.dq
```

Some examples, especially networking examples, may need a free local port and a host platform with the required socket APIs.

5.2. Building the Compiler from Source

Build from source when there is no release package for your platform, or when you want to work on the compiler itself.

5.2.1 Requirements

You need:

Tool	Notes
CMake 3.10 or newer	used to generate the native build files
GNU Make	used by the repository build wrapper
A C++23 compiler and standard library	**GCC 14 or newer**; needed for `std::print` and backtrace support
LLVM 21 development files	CMake must be able to find LLVM 21's `LLVMConfig.cmake`
LLVM support library headers	zlib, zstd, libedit, curl, libxml2, and libffi development packages

The compiler is currently developed and tested with LLVM 21. Older LLVM versions, including LLVM 18 and below, are not expected to work.

5.2.2 Select a C++ Compiler

DQ uses C++23 standard-library features such as `std::print` and stacktrace support for diagnostics and crash backtraces. Ubuntu 24.04 uses GCC 13 as the default compiler, but GCC 13's standard library is too old for these features. Check the selected compiler before configuring the build:

```
g++ --version
```

The first line should show version 14 or newer.

On Debian or Ubuntu, install GCC 14 with:

```
sudo apt update
sudo apt install gcc-14 g++-14
```

Select GCC 14 for this checkout when configuring CMake, without changing the system compiler:

```
CC=gcc-14 CXX=g++-14 cmake .
```

Use this instead of the plain `cmake .` command in the configure step below. CMake remembers the selected compiler in its cache.

5.2.3 Install LLVM 21

On Debian or Ubuntu systems, use the official LLVM APT packages from `https://apt.llvm.org/`.

Install the LLVM APT helper script and request LLVM 21:

```
sudo apt install wget ca-certificates gnupg
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 21
```

Install the packages needed to build DQ:

```
sudo apt install \
  build-essential cmake llvm-21-dev clang-21 lld-21 \
  zlib1g-dev libzstd-dev libedit-dev libcurl4-openssl-dev \
  libxml2-dev libffi-dev
```

Verify that LLVM 21 is available:

```
/usr/lib/llvm-21/bin/llvm-config --version
```

The command should print a `21.x` version.

5.2.4 Get The Source

Clone the DQ repository and enter the source tree:

```
git clone https://github.com/nvitya/dq-lang.git
cd dq-lang
```

5.2.5 Configure

From the repository root, generate the build files:

```
cmake .
```

This creates the root `Makefile` used by the repository's `GNUmakefile`. (See Troubleshooting if you experience problems here.)

5.2.6 Build

Build the compiler and helper tools:

```
make -j"$(nproc)"
```

The host build places the user-facing tools in the repository's `build` directory:

```
build/dq-comp
build/dq-run
build/dq-atrun
```

5.2.7 Install

Install the built tools and standard packages:

```
sudo make install
```

On Linux, the install target installs `dq-comp`, `dq-run`, and `dq-atrun` into `lib/dq/bin` and creates relative symlinks to them in `bin` under the chosen CMake install prefix. Standard DQ packages are installed into `lib/dq/stdpkg`. The default prefix is usually `/usr/local`.

To install under a different prefix, configure with `CMAKE_INSTALL_PREFIX`:

```
cmake . -DCMAKE_INSTALL_PREFIX="$HOME/.local"
make -j"$(nproc)"
make install
```

Then make sure the selected `bin` directory is on your `PATH`.

5.2.8 Try The Compiler

Compile and run a single DQ file with the installed tools:

```
dq-run examples/langdemo.dq
```

If you are working on your own file:

```
dq-run path/to/file.dq
```

`dq-run` uses `-g -O0` by default when no compiler options are supplied, which is useful while developing or debugging.

5.2.9 Troubleshooting

5.2.9.1 cmake .

If CMake finds another LLVM version, or cannot find LLVM at all, pass LLVM 21's CMake package directory explicitly:

```
cmake . -DLLVM_DIR=/usr/lib/llvm-21/lib/cmake/llvm
```

To build with a specific compiler without changing the system default, pass it through `CC` and `CXX` before the first configure command. For example:

```
CC=gcc-14 CXX=g++-14 cmake .
```

If configuration fails inside LLVM's CMake files with a missing imported target such as `ZLIB::ZLIB`, one of LLVM's support-library development packages is missing. Install the dependency packages listed above, then run `cmake .` again.

If configuration or compilation fails with a missing C++23 standard-library header such as `print` or `stacktrace`, CMake is using a compiler or standard library that is too old. On Ubuntu 24.04 this usually means it found GCC 13. Install GCC 14 or newer, remove the old CMake cache if necessary, and configure again with `CC=gcc-14 CXX=g++-14 cmake .`.

5.2.9.2 Link Driver

`dq-comp` uses Clang as the native linker driver when it turns generated object files into an executable. A compiler built from these instructions uses LLVM's own `clang++` from the configured LLVM installation by default.

If you are using an older build, or if you want to select another linker driver, set `DQ_LINKER_DRIVER`. Its LLVM major version must match the LLVM version used to build `dq-comp`:

```
DQ_LINKER_DRIVER=clang++-21 dq-run examples/basic/test1.dq
```

You can also export it for the current shell:

```
export DQ_LINKER_DRIVER=clang++-21
```

5.2.10 Verify the Build

Run the compiler autotests from the repository root:

```
make test
```

For a single test with detailed output:

```
build/dq-atrun --clean -c build/dq-comp autotest/tests/basic/printf.dq
```

To exercise reuse of existing module artifacts, omit `--clean`:

```
build/dq-atrun -c build/dq-comp -r autotest/tests
```

Release and Cross-Build Notes covers building tools for other hosts and producing distribution archives.

5.3. Contributor and Implementation Notes

These pages explain maintenance and implementation. They are not additional language rules. The language reference describes user-visible behavior; compiler internals can change without a source-language change.

- Building the Compiler from Source covers prerequisites, the host build, and autotests.
- Module and Interface Internals describes `.o` / `.dqm_if` generation, identity, and concurrency.
- Exceptions and Backtraces describes native unwinding, generated cleanup, and fault-recovery limitations.
- Release and Cross-Build Notes covers distribution packages and compiler builds for other hosts.
- Documentation Coverage maps the previous documentation and archived design notes to their current destinations.

5.3.1 Documentation Policy

5.3.1.1 Published Sources

`docs/` and package-local `docs/` directories are the maintained sources for the published DQ documentation. The book includes package-local API manuals through the chapter files in `docs/040-packages/`. The site tracks the compiler on the repository's `main` branch during Pre-V1 development.

Documentation has four explicitly different roles:

1. **Language Reference** records implemented behavior and is normative for the documented revision.
2. **Guides** teach normal usage and link to the reference for complete rules.
3. **Contributor Documentation** describes compiler, runtime, release, and maintenance implementation.
4. **Archive** preserves superseded specifications and design work without making it part of the current contract.

5.3.1.2 Authority

When sources disagree, use this order:

1. compiler behavior demonstrated by autotests and the standard packages;
2. the published language reference;
3. guides and examples;
4. contributor implementation descriptions;
5. archived specifications and proposals.

A disagreement between the first two is a documentation defect or an uncovered compiler defect. Resolve it by deciding the implemented intent, adding or fixing an autotest, and updating the reference in the same change.

5.3.1.3 Reference Rules

- Document only implemented behavior.
- State types, lifetime, ownership, errors, and boundary conditions explicitly.
- Keep one canonical location for each detailed rule; guides summarize and link.
- Use examples from current syntax and verify non-trivial examples against the compiler.
- Do not promise future compatibility while DQ is Pre-V1.
- Put future designs in a clearly non-normative proposal, never inline as if they were partially supported.

5.3.1.4 Guide Rules

- Optimize for the common path and readable examples.

- Avoid duplicating large tables or edge-case algorithms from the reference.
- Link to the corresponding reference near the beginning of each page.
- Update navigation and cross-references when reorganizing the book.

5.3.1.5 Contributor and Design Rules

Contributor pages must say that they describe implementation rather than the language contract. A proposal must carry a visible **Design proposal — not implemented** status and must not appear under Language Reference.

When implementation lands, migrate user-visible semantics into the reference, update relevant tests, and either rewrite the proposal as current internals or archive it.

5.3.1.6 Archival Rules

Archived files live under `doc/archive/`, outside MkDocs. Each file starts with a historical status banner and points to the coverage map or replacement page. Archived material is preserved, not maintained: corrections belong in `docs/`.

5.3.1.7 Change Checklist

- Does a language change update its reference page and a test?
- Does a new user-facing tool option update the compiler guide?
- Are guide and reference links reciprocal where both exist?
- Does `mkdocs build --strict` pass with no omitted pages or broken anchors?
- Do relevant compiler autotests pass?
- Was superseded design material archived rather than left ambiguous?

5.3.2 Writing Current DQ

This compact checklist helps contributors and AI agents produce code matching the repository. It is not another specification: follow the Language Reference and prefer working examples in `stdpkg/`, `examples/`, and `autotest/tests/` if this checklist becomes stale.

5.3.2.1 Default Style

- Use colon/end-keyword blocks and four-space indentation in documentation examples.
- Use explicit types where they clarify an example; follow the declaration rules for inference.
- Use `str` for owned text, `rostr`, `strslice` for read-only borrowed text, and `embstr(N)` only for fixed C-compatible storage.
- Use double quotes for text. A one-character literal in single quotes denotes a character; see Character and String Types.
- Use lowercase Boolean operators and uppercase integer/bitwise word operators.
- Prefer `use print` and `PrintLn` for repository examples over raw `printf`.

5.3.2.2 Ownership and Aliasing

- `[*]T` owns dynamic array storage; `[]T` borrows a view.
- `str` is managed copy-on-write; `.pchar`, `rostr`, `strslice`, slices, and `embstr` descriptors borrow storage.
- Object variables are references. Use `new` / `delete` for heap objects and `<-` for fixed storage.
- Use `name : ref T`, `refin`, `refout`, or `refnil` only when aliasing is part of the function contract.

5.3.2.3 Explicit Boundaries

- Conditions must be `bool`; integers are not truthy.

- Use `Ord` and checked helpers for character or enum ordinals.
- Cast pointers explicitly and document lifetime/ABI assumptions.
- In object methods, qualify outside value symbols or use an allowed local `use`; do not assume unrestricted module lookup.

5.3.2.4 Before Submitting

- Compile the changed example or run its closest autotest.
- Add an autotest for a new language rule or boundary case.
- Update the reference for language behavior and the guide only when common usage changes.
- Do not copy rules out of `doc/archive/` without revalidating them.

5.4. Module and Interface Internals

> Contributor implementation note: this page is not part of the DQ language contract. It describes the current compiler and may change without a source language change.

5.4.1 Artifact Pair

Each compiled module produces two independent files under `.dqbuild`:

- a native `.o` object containing generated code/data for the linker;
- a standalone `.dqm_if` file containing serialized public semantic data and freshness metadata for the DQ compiler.

The `.o` is a normal target object and does not embed the DQ interface. The compiler reads public declarations from `.dqm_if`, not from the linked executable or an object section. Earlier designs that used a `.dqm` object with an embedded interface are obsolete.

5.4.2 Interface Generation

Interface scanning parses enough of a source module to form its declarations, types, constants, attributes, imports, and reexports without requiring private code generation. `--ifgen` emits the standalone interface. A normal compile emits both artifacts.

Public bodies are serialized only when importer-side compilation requires them, such as inline assembly templates and `constexpr` expressions. A `constexpr` record stores the signature and a checked expression tree. Parameter references use their indexes in the signature, and symbol references carry the defining module's identity. The compiler also records the types and symbols it needs to expand imported expressions, including through selective imports and reexports. Ordinary function bodies remain in source/code generation and are not general interface payloads.

Source `#define` values are serialized separately from ordinary public constants. A `DQMIF_DEFINE` marker precedes the existing constant record format; loading it populates the module's definition map without adding a public module symbol. Imports merge this map into the current module's `@def` scope. Only explicit reexports carry imported definitions into subsequent interfaces; source definitions in the implementation section are not exported. Typed definitions reuse constant/type serialization, with separate linker names and exported storage for aggregates. Built-in and command-line definitions are not included. Format version 1.36 adds `constexpr` records and requires older interfaces to be regenerated.

5.4.3 Identity and Dependencies

Every interface records canonical module identity, target/ABI-relevant context, source dependencies, and format/compiler compatibility data. Full compilation also associates the interface with its matching object file.

Freshness validation checks direct source records and matching artifact metadata. An invalid, incompatible, truncated, or stale interface is regenerated rather than trusted.

5.4.4 Cycles and Child Compilation

The compiler can start interface-only child compilations to satisfy imports. Build context—target, package roots, defines, optimization, debug/LTO settings, and artifact root—is passed explicitly. Active interface dependency chains are tracked so a non-resolvable cycle produces a useful diagnostic rather than blind recursive retries.

5.4.5 Writes and Concurrency

Generated files should be written atomically and coordinated per target path so parallel compilers do not expose partial interfaces or objects. Readers validate headers and records before constructing semantic objects.

5.4.6 Format Ownership

The `.dqm_if` record format is compiler-owned and versioned. It is not a stable third-party ABI or hand-editable interchange format. Use `dq-comp --ifdump` for inspection rather than parsing it in application code.

5.5. Exceptions and Backtraces

> Contributor implementation note: source-level guarantees are defined in Exceptions and Error Handling.

5.5.1 Runtime Model

Hosted DQ exceptions are object references derived from the runtime exception base type. Polymorphic type information supports handler matching. Raising an exception transfers control through compiler-generated cleanup and handler regions.

A raise captures a backtrace before propagation. Re-raising preserves the active exception context rather than constructing an unrelated exception.

5.5.2 Generated Cleanup

The compiler tracks initialized locals that own cleanup behavior. Every exit edge from a protected scope—normal fallthrough, return, loop transfer, or unwind—runs the required cleanup exactly once in reverse construction order.

`finally` is represented on all exits from its protected region. Contributor changes must test both normal control flow and exception replacement from within cleanup/finally code.

5.5.3 Matching and Top-Level Handling

Handlers are considered in source order. Runtime type-chain checks implement base/derived matching. If none match, propagation continues.

The hosted top-level handler prints the exception message and captured frames, then terminates with failure. Signal-to-exception behavior is platform/runtime specific and must not be generalized to bare targets without an implemented runtime contract.

On all supported Linux targets (x86-64, ARM64, armhf, and riscv64), SIGSEGV is converted to `ESegFault` in the signal handler and propagates through the native unwinder. The handler unblocks SIGSEGV before throwing, so subsequent faults can also be caught.

On armhf, `sigaction` supplies the saved signal context. After capturing the backtrace, the handler advances its saved PC by a halfword so ARM EHABI's return-address lookup includes a fault at the start of an exception-table range. This adjustment runs only on the signal path.

On Windows x86-64, a vectored exception handler converts access violations to `ESegFault` and propagates them through the native MinGW SEH unwinder. Local handlers and owned-scope cleanups run before control continues after `endtry`. The recursion guard is reset before throwing, allowing subsequent faults to be caught. A thin wrapper around the MinGW personality advances the faulting frame's lookup PC by one byte so its return-address lookup includes the faulting instruction at the start of an exception-table range. The original machine context and captured backtrace are preserved. Uncaught faults reach the hosted top-level exception reporter.

LLVM cannot attach unwind edges to ordinary loads and stores. On Linux and Windows with exceptions enabled, indirect memory accesses therefore use unwindable inline assembly containing the corresponding memory instruction. This adds exception-table coverage without pointer checks, checkpoints, or helper calls on the normal path. Stack and global storage accesses retain ordinary LLVM loads and stores. Aggregate accesses are lowered field by field. The unwind edges constrain optimizations around potentially faulting accesses, as they do for potentially throwing calls.

Known limitation on Linux x86-64: LLVM 21 and 22 can shrink-wrap an optimized leaf function's stack setup into its cleanup landing pad, placing the unwind entry after that setup. A SIGSEGV unwinding through such a function can therefore fail. Local signal handling is verified at `-O0` (including the `dq-run` default); optimized signal recovery is not yet reliable. No normal-path stack setup or checkpoints are added to work around this backend issue.

Windows access-violation recovery is also verified at `-O0`, using Wine; native Windows execution and optimized fault recovery have not been validated. The opt-in `tools/test-windows.py` suite runs the shared local-fault and top-level fault tests, including reads, writes, repeated faults, cleanups, finally, non-nil protected memory, and normal scalar/aggregate accesses.

5.5.4 Verification Areas

The exception autotests cover typed matching, propagation and rethrow, finally on all control transfers, local cleanup, runtime-error catching, top-level reporting, backtraces, and supported signal handling. Any change to scope cleanup or object lifetime should run these tests in addition to its local test group.

5.6. Release and Cross-Build Notes

> Maintainer procedure: these commands build distribution artifacts and are not part of the DQ language or compiler command-line contract.

5.6.1 Release Packages

Build a compact Linux package without bundled LLVM tools:

```
make -j"$(nproc)" package-linux-release
```

Cross-build and package ARM64 Linux without bundled LLVM tools:

```
make -j"$(nproc)" package-arm64-linux
# Alias: make -j"$(nproc)" package-aarch64-linux
```

This uses the `cross-aarch64-linux` build and creates `dist/dq-<version>-arm64-linux.tar.gz`. The archive includes `dq-comp`, `dq-run`, `dq-atrun`, and the standard packages. The destination needs a 64-bit ARM Linux system, the tools' runtime dependencies, and a compatible native Clang/linker. Pass `CMAKE_EXTRA_ARGS` to select the ARM64 LLVM development libraries when cross-building, as described in the cross-compilation guide below.

Linux archives include `install.sh`. After extraction on the destination, run `sudo ./install.sh` for the default `/usr/local` installation, or `./install.sh --prefix "$HOME/.local"` for a user installation. The installer also accepts `DESTDIR` or `--destdir` for staging and requires no build tools.

The installer replaces shipped components and removes obsolete files, while preserving additional packages under `<prefix>/lib/dq/packages`. Examples, documentation, autotests, and the VS Code extension remain in the archive. To prepare a deployment tree without changing the live installation:

```
DESTDIR=/tmp/dq-root ./install.sh --prefix /usr/local
```

To test an extracted Linux release, run from the archive's root:

```
bin/dq-atrun --clean -c bin/dq-comp -r autotest/tests
```

The archive includes the test helper package, C++ fixture, and internal test executables under `autotest/bin`. Native x86 assembly tests are marked `hostarch('x64')` and are skipped on ARM64. Driver tests also need `c++`, `nm`, `llvm-readobj`, and `wasm-objdump` on `PATH`. When using versioned LLVM packages, add their tool directory to `PATH`, for example `/usr/lib/llvm-21/bin`.

On Debian 13 (Trixie), after enabling Trixie backports, install the ARM64 release's runtime and linker dependencies plus the additional test tools with:

```
sudo apt install -t trixie-backports libc++1 libc++abi1 clang-21 lld-21 llvm-21
sudo apt install g++ libbacktrace-dev zlib1g-dev wabt wasi-libc
sudo apt install -t trixie-backports libclang-rt-21-dev-wasm32
export PATH="/usr/lib/llvm-21/bin:$PATH"
```

The ARM64 release requires libc++ 21; Trixie's default libc++ 19 is insufficient. The WASI libc and wasm32 builtins are needed by the cross-target driver tests.

Build the self-contained Linux package:

```
make -j"$(nproc)" package-linux-full-release
```

Build the full Windows package from its supported cross-build environment:

```
make -j"$(nproc)" package-windows-release
```

Run the normal compiler build and autotests before producing packages. Test each package in a clean environment that does not accidentally provide unbundled compiler or runtime dependencies.

5.6.2 Clean Ubuntu Test Container

One suitable smoke-test environment is Ubuntu 24.04:

```
docker pull ubuntu:24.04
docker run --name dq-ubuntu2404 -it ubuntu:24.04 bash
apt update
apt install -y lsb-release wget gnupg software-properties-common ca-certificates sudo
apt install -y build-essential git cmake ninja-build clang lld g++
```

Keep package-production dependencies separate from dependencies expected on the destination machine. The full package should be tested without relying on a system LLVM installation.

For compiler cross-compilation prerequisites and target toolchains, see Compiler Cross-Builds.

5.6.3 Compiler Cross-Builds

These procedures build `dq-comp` itself for another host architecture. They require LLVM development libraries for that host, in addition to a cross C++ toolchain. Application cross-compilation uses an existing compiler and has different requirements.

5.6.3.1 Prerequisites: LLVM Dependency

The DQ compiler depends on LLVM (`find_package(LLVM REQUIRED CONFIG)`). To build `dq-comp` so that it runs on a target architecture (e.g., `aarch64`), you must provide the target architecture's version of the LLVM libraries.

This is a known issue on Debian/Ubuntu: installing target-architecture `-dev` packages (like `llvm-dev:arm64`) can conflict with critical host packages and try to remove your desktop environment!

Do NOT use `apt` to install target LLVM development libraries.

Instead, use the official pre-built LLVM binaries for the target architecture, or build LLVM from source for the target.

5.6.3.1.1 EXAMPLE: SETTING UP FOR AARCH64-LINUX (USING PRE-BUILT LLVM)

1. **Install the cross-compiler and multiarch dependencies on the host:** Because the official LLVM binaries for Linux are compiled with `libc++` and Link Time Optimization (LTO), we must use `clang++` and `lld` as the cross-compiler instead of GNU `g++` . We also need the target architecture's `libc++` , `zlib` , `zstd` , and `libxml2` headers. `bash`
   ```
   sudo apt update
   sudo apt install clang lld \
        libc++-dev:arm64 libc++abi-dev:arm64 \
        zlib1g-dev:arm64 libzstd-dev:arm64 libxml2-dev:arm64
   ```
2. **Download Pre-built LLVM for aarch64:** Go to the LLVM GitHub Releases page and download the `aarch64-linux-gnu` tarball for the matching LLVM version (LLVM 21 for this compiler).

```
bash
   mkdir -p sysroots/llvm-aarch64
   cd sysroots
   wget https://github.com/llvm/llvm-project/releases/download/llvmorg-21.1.0/LLVM-21.1.0-Linux-ARM64.tar.xz
   tar -xf LLVM-21.1.0-Linux-ARM64.tar.xz -C llvm-aarch64 --strip-components=1
   cd ..
```

1. **Configure CMake to use the downloaded LLVM:** When running the make target, you can pass the `LLVM_DIR` directly via the environment or add it to the make command. For our setup, we can explicitly tell CMake where to find it:

```
bash
   make cross-aarch64-linux CMAKE_EXTRA_ARGS="-DLLVM_DIR=$(pwd)/sysroots/llvm-aarch64/lib/cmake/llvm"
```

(For other architectures like `armhf` or `riscv64` , substitute `:arm64` with `:armhf` or `:riscv64` in the `apt install` command above.)

5.6.3.2 Building

We have added specific Make targets that simplify the CMake configuration for cross-compiling. The toolchain files are located in `toolchains/`.

5.6.3.2.1 SUPPORTED TARGETS

- `make cross-aarch64-linux`
- `make cross-armhf-linux`
- `make cross-rv64g-linux`
- `make cross-x86_64-win`

5.6.3.3 Windows x86_64

The Windows cross-build is different from the Linux cross-builds in two ways:

1. You need a Windows cross C/C++ toolchain that runs on the Linux host.
2. `dq-comp` itself links against LLVM C++ libraries, so the build still needs Windows-target LLVM development libraries visible through `LLVM_DIR`.

The `llvm-mingw` project is a good source for the first part. For an x86_64 Linux host building x86_64 Windows binaries, use the Ubuntu x86_64 tarball:

```
llvm-mingw-20251216-ucrt-ubuntu-22.04-x86_64.tar.xz
```

For the self-contained Windows DQ package, the matching Windows-hosted toolchain archive is also required:

```
llvm-mingw-20251216-ucrt-x86_64.zip
```

The `.tar.xz` archive is used on the Linux build machine. The `.zip` archive is for the final Windows package, where it can provide `clang.exe`, `lld`, startup objects, CRT libraries, and Windows import libraries for DQ users.

5.6.3.3.1 RECOMMENDED LOCAL LAYOUT

Keep checked-in CMake toolchain files in `toolchains/`. Put downloaded and extracted binary toolchains under `sysroots/`, which is ignored by git:

```
mkdir -p sysroots/llvm-mingw-x86_64
tar -xf toolchains/llvm-mingw-20251216-ucrt-ubuntu-22.04-x86_64.tar.xz \
  -C sysroots/llvm-mingw-x86_64 --strip-components=1
```

If the archive was downloaded somewhere else, use that path instead of the `toolchains/...tar.xz` path above. The important result is:

```
sysroots/llvm-mingw-x86_64/bin/x86_64-w64-mingw32-clang++
sysroots/llvm-mingw-x86_64/x86_64-w64-mingw32/
```

5.6.3.3.2 CONFIGURE WITH LLVM-MINGW

The Windows toolchain file accepts `DQ_MINGW_ROOT`:

```
make cross-x86_64-win CMAKE_EXTRA_ARGS="\
  -DDQ_MINGW_ROOT=$(pwd)/sysroots/llvm-mingw-x86_64 \
  -DLLVM_DIR=/path/to/windows-target-llvm/lib/cmake/llvm"
```

`DQ_MINGW_ROOT` points CMake at the llvm-mingw cross compiler and target headers/libraries. `LLVM_DIR` must point at the LLVM CMake package for LLVM libraries that are linkable into the Windows `dq-comp.exe`.

5.6.3.3.3 IMPORTANT LIMITATION

The llvm-mingw release archive provides the compiler, linker, CRT/startup objects, and Windows import libraries. It does not provide the LLVM development CMake package used by this project:

```
lib/cmake/llvm/LLVMConfig.cmake
```

Therefore llvm-mingw alone is probably not sufficient to build `dq-comp.exe`. You still need a compatible Windows-target LLVM development build. The most reliable approach is to build LLVM for the same MinGW/UCRT ABI using the llvm-mingw toolchain, then pass that build's `lib/cmake/llvm` directory as `LLVM_DIR`.

5.6.3.3.4 BUILDING THE WINDOWS LLVM DEVELOPMENT PACKAGE

Use the helper script after extracting the llvm-mingw Linux-hosted toolchain:

```
tools/build-llvm-mingw-ucrt.sh
```

The script builds and installs a minimal LLVM development package here:

```
sysroots/llvm-x86_64-win-ucrt
```

It downloads the matching LLVM source archive if needed, configures LLVM for `x86_64-w64-windows-gnu`, builds the X86, ARM, RISC-V, and WebAssembly target backends, disables optional compression/XML/terminal dependencies, and installs the CMake package needed by `dq-comp`:

```
sysroots/llvm-x86_64-win-ucrt/lib/cmake/llvm/LLVMConfig.cmake
```

After that, configure the DQ compiler cross-build with:

```
make cross-x86_64-win CMAKE_EXTRA_ARGS="\
  -DDQ_MINGW_ROOT=$(pwd)/sysroots/llvm-mingw-x86_64 \
  -DLLVM_DIR=$(pwd)/sysroots/llvm-x86_64-win-ucrt/lib/cmake/llvm"
```

5.6.3.3.5 BUILDING A WINDOWS RELEASE PACKAGE

After the cross-build succeeds, create a distributable zip with:

```
make package-x86_64-win CMAKE_EXTRA_ARGS="\
  -DDQ_MINGW_ROOT=$(pwd)/sysroots/llvm-mingw-x86_64 \
  -DLLVM_DIR=$(pwd)/sysroots/llvm-x86_64-win-ucrt/lib/cmake/llvm"
```

The package is written to:

```
dist/dq-<version>-x86_64-windows-ucrt.zip
```

It contains:

- `bin/dq-comp.exe`
- `bin/dq-run.exe`
- `bin/dq-atrun.exe`
- the required llvm-mingw runtime DLLs
- the Windows-hosted llvm-mingw linker SDK under `toolchain/llvm-mingw/`
- `stdpkg/`
- project license and Windows quick-start notes

The package script expects the Windows-hosted llvm-mingw archive here by default:

```
sysroots/llvm-mingw-20251216-ucrt-x86_64.zip
```

Override it with `DQ_WINDOWS_TOOLCHAIN_ZIP=/path/to/llvm-mingw-ucrt-x86_64.zip`, or use an already extracted Windows-hosted toolchain with:

```
DQ_WINDOWS_TOOLCHAIN_ROOT=/path/to/llvm-mingw-20251216-ucrt-x86_64 \
  tools/package-windows-release.sh
```

At runtime, `dq-comp.exe` links application executables with the packaged toolchain:

```
toolchain\llvm-mingw\bin\clang++.exe --target=x86_64-w64-windows-gnu -fuse-ld=lld
```

The compiler locates this driver relative to `dq-comp.exe`, so only the package's `bin` directory needs to be added to `PATH`. Keep the package directories together. Set `DQ_LINKER_DRIVER` to override that driver path.

If the helper needs a different native `llvm-tblgen`, set it explicitly:

```
LLVM_TBLGEN=/path/to/llvm-tblgen tools/build-llvm-mingw-ucrt.sh
```

Prebuilt package managers such as MSYS2 may also provide `LLVMConfig.cmake`, but mixing their LLVM libraries with the llvm-mingw toolchain can create C++ ABI and runtime mismatches. For the first Windows DQ package, prefer building LLVM with the same llvm-mingw toolchain used to build `dq-comp.exe`.

5.6.3.3.6 CURRENT PORTABILITY STATUS

The repository has a Windows cross-build target and the hosted compiler tools now build as Windows executables with the llvm-mingw toolchain:

- `dq-comp.exe`
- `dq-run.exe`
- `dq-atrun.exe`

The compiler sources now guard Linux-only backtrace support, use portable executable discovery, and provide a Windows implementation of the process runner and artifact locking. Application linking uses the bundled llvm-mingw toolchain, and hosted Windows programs use a minimal `rtl/rtl_windows.dq` runtime entry point. On Windows x86-64, access violations are converted to catchable `ESegFault` exceptions through native SEH. Fault recovery is tested under Wine at `-O0`; see Exceptions and Backtraces for implementation details and optimization limitations.

5.6.3.3.7 EXAMPLE: BUILDING FOR AARCH64

From the root of the `dq-lang` project, run:

```
make cross-aarch64-linux
```

This will: 1. Create a build directory at `build-cross/aarch64-linux`. 2. Run CMake with the repository's `toolchains/aarch64-linux.cmake` toolchain file. 3. Compile the project for the `aarch64` target.

The tools are written to `build-cross/aarch64-linux/bin/`, including `dq-comp`, `dq-run`, and `dq-atrun`. They run on the target host, rather than on the build machine.

5.6.3.4 Toolchain Files

If you need to customize the cross-compilation environment (for example, setting a custom sysroot or pointing to a locally compiled LLVM), you can edit the corresponding toolchain file in the `toolchains/` directory.

- `CMAKE_FIND_ROOT_PATH` is currently pointed to the default Ubuntu cross-compiler sysroot (e.g., `/usr/aarch64-linux-gnu`).

5.7. Glossary and Index

Use the site's search for individual declarations and methods. This index links common terms to their detailed descriptions and also works in the PDF book.

Term	Meaning and reference
Addressable value	Storage whose address can be taken with `%`; see Pointers.
`anyvalue`	A tagged value for supported scalars and text; see `anyvalue`.
Attribute	Declaration metadata written `[[...]]`; see Attributes.
`autofree`	An owner that releases its object or pointer; see Memory Management and Autofree.
Borrowed view	A reference to storage whose lifetime is managed elsewhere; see Arrays and Slices and Character and String Types.
C ABI	Native calling and layout conventions used by external declarations; see C Interoperability.
Canonical module identity	A module's identity within its module/package root; see Modules and Packages.
Copy-on-write	Shared string storage becomes independent when modified; see Character and String Types.
Directive	A compiler instruction such as `#if` or `#include`; see Compiler Directives.
Dynamic array	Managed `[*]T` storage shared by assignments; see Arrays and Slices.
Embedded object	An object stored directly in local storage, the global data segment, or a containing object, declared with `<-`; see Objects and Properties.
Embedded string	A bounded string descriptor or fixed `embstr(N)` storage; see Character and String Types.
Enumeration	A named set of ordinal values; see Enumerations.
Exception	A failure propagated to a matching handler; see Exceptions and Error Handling.
Function reference	A callable value, optionally bound to an object; see Functions.
Interface file	Compiler-owned `.dqm_if` public semantic data; see Build Process and Module Artifacts.
LTO	Link-time optimization across compiled modules; see Compiler Usage and Options.
Module	One `.dq` source file with public and implementation regions; see Modules and Packages.
Namespace	A qualified scope reached with `@name`; see Modules and Packages.
Ownership	Responsibility for releasing an allocation; see Memory Management and Autofree.
Package	A source-module tree found through package roots; see Library Overview.
Prelude	The implicit `rtl/sys` interface; see `sys`.
Project file	A `.dqproj` collection of compiler/build settings; see Project Files.
Property	Field-like syntax backed by a field or accessor; see Objects and Properties.
Reference parameter	An alias supplied with `ref`, `refin`, `refout`, or `refnil`; see Functions.
Reexport	Imported declarations exposed by another module; see Modules and Packages.
Runtime	DQ support for startup, managed storage, and checks; see Runtime and Program Execution.
Slice	A borrowed view of array or text storage; see Arrays and Slices and Character and String Types.
Structure / union	Direct aggregate storage; see Structures and Unions.

Term	Meaning and reference
Sysroot	Target headers and native libraries for a cross build; see Targets and Cross-Compilation.
Target	Machine, ABI, runtime, and feature defaults; see Targets and Cross-Compilation.
Virtual dispatch	Method selection using an object's runtime type; see Objects and Properties.

5.8. Documentation Coverage

The maintained book lives in `docs/`, organized into Project, Language, Compiler and Tools, Standard Packages, and Appendices. The previous site's sources remain under `doc/docs-v1/` as a migration snapshot. Design history remains under `doc/archive/`; neither location overrides the current book.

5.8.1 Previous Site to Current Book

Guide and reference material for the same feature is combined into one language chapter. The later language chapters retain the material migrated before this completion pass. Exceptions have a dedicated chapter, while attributes, directives, and C interoperability are separate topics.

Previous page under `doc/docs-v1/docs/`	**Current destination**
`index.md`	DQ-lang Project Introduction
`motivation.md`	Features and Design Goals
`getting-started.md`	Getting Started
`reference/index.md`	DQ Programming Language
`language/basics.md`	Basic Syntax
`reference/lexical-structure.md`	Basic Syntax
`language/types.md`	Basic Types and Conversions
`reference/types-and-conversions.md`	Basic Types and Conversions
`reference/declarations-and-scope.md`	Variables, Constants, References, and Scope
`language/expressions.md`	Expressions and Operators
`reference/expressions-and-operators.md`	Expressions and Operators
`language/statements.md`	Statements and Control Flow
`reference/statements-and-control-flow.md`	Statements and Control Flow
`language/functions.md`	Functions
`reference/functions.md`	Functions
`reference/strings-and-characters.md`	Character and String Types
`rtl/strings.md`	String Functions
`rtl/textfmt.md`	Text Formatting
`reference/arrays-and-slices.md`	Arrays and Slices
`reference/enums.md`	Enumerations
`reference/structures-pointers-and-function-references.md`	Structures and Unions
`language/memory-and-pointers.md`	Pointers
`language/objects.md`	Objects and Properties
`reference/objects-and-properties.md`	Objects and Properties
`reference/autofree.md`	Memory Management and Autofree
`reference/anyvalue.md`	anyvalue

Previous page under `doc/docs-v1/docs/`	Current destination
`rtl/anyvalue.md`	anyvalue
`language/modules.md`	Modules and Packages
`reference/modules-and-packages.md`	Modules and Packages
`language/attributes-and-directives.md`	Attributes
`reference/attributes-directives-and-interop.md`	Attributes
`language/interop.md`	C Interoperability
`language/asm-functions.md`	Assembly Functions
`reference/assembly-functions.md`	Assembly Functions
`compiler/using-dq-comp.md`	Compiler Usage and Options
`compiler/project-file.md`	Project Files (`.dqproj`)
`compiler/dqm-format.md`	Build Process and Module Artifacts
`compiler/linux-devenv.md`	Linux Development Environment
`compiler/windows-devenv.md`	Windows Development Environment
`compiler/vscode-setup.md`	VSCode Setup and Debugging
`compiler/windows-debugging.md`	VSCode Setup and Debugging
`rtl/overview.md`	Library Overview
`examples/index.md`	Example Programs
`compiler/getting-dq-comp.md`	Building the Compiler from Source
`contributing/index.md`	Contributor and Implementation Notes
`contributing/documentation-policy.md`	Contributor and Implementation Notes
`contributing/writing-current-dq.md`	Contributor and Implementation Notes
`contributing/module-and-interface-internals.md`	Module and Interface Internals
`contributing/exceptions-and-backtraces.md`	Exceptions and Backtraces
`compiler/cross-compiling.md`	Release and Cross-Build Notes
`contributing/release-and-cross-builds.md`	Release and Cross-Build Notes
`contributing/coverage-inventory.md`	Documentation Coverage

The former combined structure/pointer/function-reference page is split among Structures and Unions, Pointers, and Functions. The old statement pages' exception rules are in Exceptions and Error Handling. The combined attributes page is split among Attributes, Compiler Directives, and C Interoperability.

The old compiler usage page is split among the tool overview, option reference, artifact chapter, and application targets. Compiler cross-build procedures belong to the maintainer appendix. Release installation is covered by the Linux and Windows environment chapters; source builds have their own appendix.

The old runtime overview is expanded into Runtime and Program Execution, Library Overview, and `sys`. There is also a dedicated `print` chapter.

5.8.2 Package Manuals

Package-local documentation remains the maintained API source. The book includes these files directly, so a correction is published in both forms.

Source	Book chapter
`stdpkg/file/docs/index.md`	`file`
`stdpkg/json/docs/index.md`	`json`
`stdpkg/threads/docs/index.md`	`threads`
`stdpkg/time/docs/index.md`	`time`
`stdpkg/nanonet/docs/index.md`	`nanonet`
`stdpkg/shell/docs/index.md`	`shell`

5.8.3 Archived Design Coverage

This inventory records the disposition of the former `doc/` specifications. The archived copies are historical and do not override the listed destinations.

Archived source	Disposition and canonical current documentation
Original language specification and changelog	Historical baseline. Replaced by the complete Language Reference.
Character/integer conversion specification	Migrated to Types and Conversions and Strings and Characters.
Enum specification	Migrated to Enumerations.
Array specification	Migrated to Arrays and Slices, including the array methods.
String specification and String Handling v2	Reconciled in Strings and Characters and String Methods. Conflicting earlier character models are historical.
`anyvalue` specification	Public behavior and runtime methods moved to `anyvalue`. Private layout remains compiler/runtime-owned.
Object allocation/lifetime specification	Migrated to Objects and Properties.
Property specification	Migrated to Objects and Properties.
Object type-information notes	User-visible tests/casts moved to Objects and Properties; implementation detail belongs to compiler code.
Exception/backtrace specification	Semantics moved to Exceptions and Error Handling; implementation summarized in Exceptions and Backtraces.
Module system specification	Source behavior migrated to Modules and Packages.
Package specification	Search and identity behavior migrated to Modules and Packages and Using the Compiler.
Module implementation notes	Current design summarized in Module and Interface Internals; obsolete embedded-interface designs remain historical.
DQM interface-format specification	Current user contract is DQ Module Artifacts; implementation is summarized in Module and Interface Internals.
Build specification	Current user behavior is Using the Compiler; superseded `.dqm` layouts are historical.

Archived source	Disposition and canonical current documentation
Cross-compiling notes	Maintained user workflow is Cross-Compiling and Packaging; maintainer notes are Release and Cross-Build Notes.
Method-local `use` notes	Migrated to Declarations and Scope and Modules and Packages.
DQ writing hints for AI agents	Replaced by Writing Current DQ, which links to normative pages rather than duplicating specifications.

5.8.4 Original Specification Sections

Original area	Canonical destination
Overview and design principles	Home, Motivation
Lexical structure and literals	Lexical Structure
Types, variables, and constants	Types and Conversions, Declarations and Scope
Expressions and operators	Expressions and Operators
Statements and control flow	Statements and Control Flow
Functions	Functions
Objects and properties	Objects and Properties
Pointers and memory	Structures, Pointers, and Function References, Objects and Properties
Modules and namespaces	Modules and Packages
Compiler directives and C access	Attributes, Directives, and Interoperability
Standard library	Runtime Library and Standard Packages sections of the site
Rejected features and open questions	Historical only unless reintroduced as a separate design proposal