Daslang Reference Manual

Release 0.6.0

Boris Batkin

Feb 27, 2026

CONTENTS

CHAPTER

ONE

INTRODUCTION

Daslang is a high-performance, strong and statically typed scripting language, designed to be high-performance as an embeddable "scripting" language for real-time applications (like games).

Daslang offers a wide range of features like strong static typing, generic programming with iterative type inference, Ruby-like blocks, semantic indenting, native machine types, ahead-of-time "compilation" to C++, and fast and simplified bindings to C++ program.

It's philosophy is build around a modified Zen of Python.

- *Performance counts.*
- *But not at the cost of safety.*
- *Unless is explicitly unsafe to be performant.*
- *Readability counts.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Flat is better than nested.*

Daslang is supposed to work as a "host data processor". While it is technically possible to maintain persistent state within a script context (with a certain option set), Daslang is designed to transform your host (C++) data/implement scripted behaviors.

In a certain sense, it is pure functional - i.e. all persistent state is out of the scope of the scripting context, and the script's state is temporal by its nature. Thus, the memory model and management of persistent state are the responsibility of the application. This leads to an extremely simple and fast memory model in Daslang itself.

1.1 Performance.

In a real world scenarios, it's interpretation is 10+ times faster than LuaJIT without JIT (and can be even faster than LuaJIT with JIT). Even more important for embedded scripting languages, its interop with C++ is extremely fast (both-ways), an order of magnitude faster than most other popular scripting languages. Fast calls from C++ to Daslang allow you to use Daslang for simple stored procedures, and makes it an ECS/Data Oriented Design friendly language. Fast calls to C++ from Daslang allow you to write performant scripts which are processing host (C++) data, and rely on bound host (C++) functions.

It also allows Ahead-of-Time compilation, which is not only possible on all platforms (unlike JIT), but also always faster/not-slower (JIT is known to sometimes slow down scripts).

Daslang already has implemented AoT (C++ transpiler) which produces code more or less similar with C++11 performance of the same program.

Table with performance comparisons on a synthetic samples/benchmarks.

1.2 How it looks?

Mandatory fibonacci samples:

```
def fibR(n) {
    if (n < 2) {
        return n
    } else {
        return fibR(n - 1) + fibR(n - 2)
    }
}

def fibI(n) {
    var last = 0
    var cur = 1
    for ( i in 0..n-1 ) {
        let tmp = cur
        cur += last
        last = tmp
    }
    return cur
}
```

The same samples with significant white space (python style), for those who prefer this type of syntax:

```
options gen2=false

def fibR(n)
   if n < 2
       return n
   else
       return fibR(n - 1) + fibR(n - 2)

def fibI(n)
   var last = 0
   var cur = 1
   for i in 0 .. n-1
       let tmp = cur
       cur += last
       last = tmp
   return cur
```

At this point gen2 style syntax (with curly bracers) is the default, but you can switch to gen1 style (with indentation) by setting the *gen2* option to *false*.

1.3 Generic programming and type system

Although above sample may seem to be dynamically typed, it is actually generic programming. The actual instance of the fibI/fibR functions is strongly typed and basically is just accepting and returning an `int`. This is similar to templates in C++ (although C++ is not a strong-typed language) or ML. Generic programming in Daslang allows very powerful compile-time type reflection mechanisms, significantly simplifying writing optimal and clear code. Unlike C++ with it's SFINAE, you can use common conditionals (if) in order to change the instance of the function depending on type info of its arguments. Consider the following example:

```
def setSomeField(var obj; val) {
    static_if ( typeinfo has_field<someField>(obj) ) {
        obj.someField = val
    }
}
```

This function sets *someField* in the provided argument *if* it is a struct with a *someField* member.

(For more info, see *Generic programming*).

1.4 Compilation time macros

Daslang does a lot of heavy lifting during compilation time so that it does not have to do it at run time. In fact, the Daslang compiler runs the Daslang interpreter for each module and has the entire AST available to it.

The following example modifies function calls at compilation time to add a precomputed hash of a constant string argument:

```
[tag_function_macro(tag="get_hint_tag")]
class GetHintFnMacro : AstFunctionAnnotation {
    def override transform(var call : smart_ptr<ExprCallFunc>; var errors : das_string)␣
↪: ExpressionPtr {
        if (call.arguments[1] is ExprConstString) {
            unsafe {
                var new_call := call // <- clone_expression(call)
                let arg2 = reinterpret<ExprConstString?>(call.arguments[1])
                let hint = hash("{arg2.value}")
                emplace_new(new_call.arguments, new ExprConstUInt64(at = arg2.at, value␣
↪= hint))
                return new_call
            }
        }
        return <- default<ExpressionPtr>
    }
}
```

1.5 Features

Its (not) full list of features includes:

- strong typing
- Ruby-like blocks and lambda
- tables
- arrays
- string-builder
- native (C++ friendly) interop
- generics
- classes
- macros, including reader macros
- semantic indenting
- ECS-friendly interop
- easy-to-extend type system
- etc.

CHAPTER

TWO

THE LANGUAGE

2.1 Program Structure

A daslang source file is a sequence of top-level declarations. This page describes the overall layout of a file and the key declarations that control how it interacts with the rest of the program.

2.1.1 File Layout

A typical daslang file follows this layout:

```
options gen2                                // compilation options

module my_module shared public              // module declaration

require math                                // imports
require daslib/strings_boost

struct MyData                               // type declarations
    value : int
    name : string

enum Color {
    red
    green
    blue
}

let MAX_COUNT = 100                         // global constants

var total : int = 0                         // global variables

typedef IntArray = array<int>               // type aliases

def helper(x : int) : int {                 // functions
    return x * 2
}

[export]                                    // entry point
def main {
```

(continues on next page)

(continued from previous page)

```
    print("hello\n")
}
```

The parser does not enforce a strict ordering among `options`, `module`, and `require`. However, the `module` declaration must appear before any type declarations (structs, enums, functions, global variables, type aliases). By convention, `options` lines come first, followed by `module`, then `require`.

Note: The order above is a convention, not a hard rule. The only enforced constraint is that `module` precedes type declarations.

2.1.2 Module Declaration

The `module` declaration names the current file's module:

```
module my_module
```

If omitted, the module name defaults to the file name (without extension).

Modifiers

The `module` declaration supports several modifiers:

`shared`
: Promotes the module to a built-in module. Only one instance is created per compilation environment, and it is shared across contexts:

```
module my_lib shared
```

`public` / `private`
: Sets the default visibility of all declarations in the module. Functions, structs, enums, and globals inherit this default unless they specify their own visibility:

```
module my_lib public          // all declarations are public by default
module my_lib private         // all declarations are private by default
```

If neither is specified, the module uses the environment's default (typically `public`).

`inscope`
: Makes the module visible to all modules in the project without an explicit `require`. This uses the `!inscope` syntax:

```
module my_lib !inscope
```

Modifiers can be combined:

```
module my_lib shared public
```

2.1.3 Require Declaration

The `require` declaration imports another module:

```
require math
require daslib/ast_boost
```

Module names can contain `/` and `.` separators. The project is responsible for resolving module names into file paths.

Re-exporting

By default, required modules are private — they are only visible within the current module. The `public` modifier re-exports the module, making it transitively visible to any module that requires the current one:

```
require dastest/testing_boost public
```

Aliasing

When two modules share the same name, the `as` keyword provides a local alias:

```
require event
require sub/event as sub_event

def handle {
    sub_event::process()          // qualified call using the alias
}
```

(see *Modules* for details on module function visibility and the `_` / `__` module prefixes).

2.1.4 Options Declaration

The `options` declaration sets compiler options for the file:

```
options gen2
options no_unused_block_arguments = false
```

Multiple options can appear on one line, separated by commas:

```
options no_aot = true, rtti = true
```

A bare option name (without `= value`) is shorthand for `= true`:

```
options gen2          // equivalent to: options gen2 = true
```

(see *Options* for the complete list of recognized options).

2.1.5 Top-Level Declarations

After the header declarations, the rest of the file consists of:

- **Type aliases** — `typedef`, named `tuple`, `variant`, `bitfield` (see *Type Aliases*)
- **Enumerations** — `enum` (see *Constants and Enumerations*)
- **Structures and classes** — `struct`, `class` (see *Structs*, *Classes*)
- **Global variables** — `let` (constant) and `var` (mutable) (see *Constants and Enumerations*)
- **Functions** — `def` (see *Functions*)
- **Top-level reader macros** — `%macro_name~...~`

All of these are peers in the grammar and can appear in any order, interleaved freely.

2.1.6 Visibility

Each top-level declaration can be marked `public` or `private`:

```
def public helper(x : int) : int {     // visible to other modules
    return x * 2
}

struct private Internal {               // only visible within this module
    data : int
}
```

If no visibility is specified, the declaration inherits the module's default visibility.

Shared global variables use the `shared` keyword and are shared across cloned contexts:

```
let shared GLOBAL_TABLE : table<string; int>
```

2.1.7 Entry Points

A daslang program is compiled and simulated by the host application (a C++ executable). The host decides which functions to call. Several annotations mark functions with special roles.

[export]

Marks a function as callable from the host application. The host invokes exported functions by name through the context API:

```
[export]
def main {
    print("hello world\n")
}
```

There is nothing special about the name `main` — it is purely a convention. The host chooses which exported function(s) to call and in what order.

[init]

Marks a function to run automatically during context initialization. `[init]` functions cannot have arguments and cannot return a value:

```
[init]
def setup {
    print("initializing\n")
}
```

Multiple `[init]` functions execute in declaration order. Ordering can be controlled with attributes:

```
[init(tag="db")]
def init_database {
    pass
}

[init(after="db")]
def init_cache {
    pass
}

[init(before="db")]
def init_logging {
    pass
}
```

The option `no_init` disables all `[init]` functions.

[finalize]

Marks a function to run automatically during context shutdown. Same constraints as `[init]` — no arguments, no return value:

```
[finalize]
def cleanup {
    print("shutting down\n")
}
```

2.1.8 Expect Declaration

The `expect` declaration is used in test files to declare expected compilation errors. When present, the compiler treats the listed errors as intentional — the file compiles "successfully" only if exactly those errors (and no others) are produced.

This is primarily used in negative test suites to verify that the compiler correctly rejects invalid code:

```
expect 40214:3              // expect error 40214 exactly 3 times
expect 30304, 30101         // expect each error once (count defaults to 1)
```

The syntax is:

```
expect <error_code> [: <count>] [, <error_code> [: <count>] ...]
```

Multiple `expect` declarations can appear in the same file. Error codes are numeric identifiers organized by compilation phase:

Range	Category
`10001-10011`	Lexer errors (mismatched brackets, etc.)
`20000-20001`	Parser errors (syntax errors)
`30101-30128`	Semantic: invalid type/annotation/name
`30201-30213`	Semantic: already declared / too many args
`30301-30311`	Semantic: not found (type, func, var, etc.)
`30401-30403`	Semantic: can't initialize
`30501-30509`	Semantic: can't dereference/copy/move
`30601-30602`	Semantic: condition errors
`31300`	Unsafe operation outside `unsafe` block
`39901-39903`	Semantic: missing value/typeinfo
`40101-40214`	Lint-time errors and warnings

For example, a test that verifies the compiler rejects copying an array:

```
expect 30507    // cant_copy

[export]
def main {
    var a <- [1, 2, 3]
    var b = a           // error: can't copy array
}
```

2.1.9 Program vs. Module

The same file format is used for both programs and modules. The distinction is how the file is used:

Program (entry point)
: The top-level file compiled by the host application. Typically contains `[export]` functions. May omit the `module` declaration.

Module (library)
: A file imported via `require` by other files. Typically has a `module` declaration and provides types, functions, and globals for reuse.

A file can serve both roles simultaneously.

2.1.10 Execution Lifecycle

1. The host compiles a source file into a `Program`
2. The program is simulated into a `Context`
3. The context is initialized:
   - Global variables are initialized in declaration order, per module
   - `[init]` functions run in declaration order (or topologically, if ordering attributes are used)
4. The host calls `[export]` functions as needed
5. The context is shut down:

- `[finalize]` functions run

2.1.11 Complete Example

The following example shows a complete program with all structural elements:

```
options gen2

require math
require daslib/strings_boost

struct Particle {
    pos : float3
    vel : float3
    life : float
}

enum State {
    alive
    dead
}

let GRAVITY = float3(0.0, -9.8, 0.0)

var particles : array<Particle>

def update_particle(var p : Particle; dt : float) : State {
    p.vel += GRAVITY * dt
    p.pos += p.vel * dt
    p.life -= dt
    if (p.life > 0.0) {
        return State.alive
    }
    return State.dead
}

[init]
def setup {
    for (i in range(100)) {
        particles |> push(Particle(pos=float3(0), vel=float3(0, 10.0, 0), life=5.0))
    }
}

[export]
def main {
    let dt = 0.016
    for (p in particles) {
        update_particle(p, dt)
    }
    print("particles: {length(particles)}\n")
}

[finalize]
```

(continues on next page)

(continued from previous page)

```
def cleanup {
    unsafe {
        delete particles
    }
}
```

Expected output:

```
particles: 100
```

See also:

Modules for module declaration and `require` semantics, *Options* for compiler and runtime options, *Annotations* for `[init]`, `[finalize]`, and `[export]`, *Contexts* for the execution context lifecycle, *Constants and enumerations* for global declarations.

2.2 Lexical Structure

2.2.1 Identifiers

Identifiers start with an alphabetic character or an underscore (_), followed by any number of alphabetic characters, underscores, digits (`0`–`9`), or backticks (`` ` ``). Daslang is a case-sensitive language, meaning that `foo`, `Foo`, and `fOo` are three distinct identifiers.

Backticks are used in system-generated identifiers (such as mangled names) and are generally not used in user code:

```
my_variable     // valid
_temp           // valid
player2         // valid
```

2.2.2 Keywords

The following words are reserved as keywords and cannot be used as identifiers:

struct	*class*	*let*	*def*	*while*	*if*
static_if	`else`	*for*	`recover`	`true`	`false`
`new`	*typeinfo*	`type`	`in`	*is*	*as*
`elif`	`static_elif`	*array*	*return*	`null`	*break*
`try`	*options*	*table*	`expect`	`const`	*require*
`operator`	*enum*	*finally*	`delete`	*deref*	*aka*
typedef	`with`	*cast*	`override`	`abstract`	*upcast*
iterator	*var*	*addr*	*continue*	`where`	*pass*
`reinterpret`	*module*	`public`	`label`	`goto`	`implicit`
`shared`	`private`	`smart_ptr`	*generator*	*yield*	*unsafe*
assume	`explicit`	`sealed`	`static`	*inscope*	`fixed_array`
`typedecl`	`capture`	`default`	`uninitialized`		`template`

The following words are reserved as built-in type names and cannot be used as identifiers:

bool	void	string	auto	int	int2
int3	int4	uint	*bitfield*	uint2	uint3
uint4	float	float2	float3	float4	range
urange	*block*	int64	uint64	double	function
lambda	int8	uint8	int16	uint16	*tuple*
variant	range64	urange64			

Keywords and types are covered in detail in subsequent sections of this documentation. Linked keywords above lead to their relevant documentation pages.

2.2.3 Operators

Daslang recognizes the following operators:

+=	-=	/=	*=	%=	|=	^=	<<
>>	++	--	<=	<<=	>>=	>=	==
!=	->	<-	??	?.	?[	<|	|>
:=	<<<	>>>	<<<=	>>>=	=>	+	@@
-	*	/	%	&	|	^	>
<	!	~	&&	||	^^	&&=	||=
^^=	..						

Notable operators unique to Daslang:

- <- — move assignment
- := — clone assignment
- ?? — null coalescing
- ?. and ?[— null-safe navigation
- <| and |> — pipe operators
- @@ — function pointer / local function
- <<< and >>> — bit rotation
- ^^ — logical exclusive or
- .. — interval (range creation)
- => — tuple construction (`a => b` creates `tuple<auto,auto>(a, b)`; also used in table literals)

2.2.4 Other Tokens

Other significant tokens are:

{	}	[	]	.	:
::	'	;	"	@	$
#					

2.2.5 Literals

Daslang accepts integer numbers, unsigned integers, floating-point and double-precision numbers, and string literals.

Numeric Literals

`34`	Integer (base 10)
`0xFF00A120`	Unsigned integer (base 16)
`075`	Unsigned integer (base 8)
`13l`	64-bit integer (base 10)
`0xFF00A120ul`	64-bit unsigned integer (base 16)
`32u8`	8-bit unsigned integer
`'a'`	Character literal (integer value)
`1.52`	Floating-point number
`1.0f`	Floating-point number (explicit suffix)
`1.e2`	Floating-point number (scientific)
`1.e-2`	Floating-point number (scientific)
`1.52d`	Double-precision number
`1.e2lf`	Double-precision number
`1.e-2d`	Double-precision number

Integer suffixes:

- `u` or `U` — unsigned 32-bit integer
- `l` or `L` — signed 64-bit integer
- `ul` or `UL` — unsigned 64-bit integer
- `u8` or `U8` — unsigned 8-bit integer

Float suffixes:

- `f` (or no suffix after a decimal point) — 32-bit float
- `d`, `lf` — 64-bit double

String Literals

Strings are delimited by double quotation marks ("). They support escape sequences and string interpolation:

```
"I'm a string\n"
```

Escape sequences:

\t	Tab
\n	Newline
\r	Carriage return
\\	Backslash
\"	Double quote
\'	Single quote
\0	Null character
\a	Bell
\b	Backspace
\f	Form feed
\v	Vertical tab
\xHH	Hexadecimal byte
\uHHHH	Unicode code point (16-bit)
\UHHHHHHHH	Unicode code point (32-bit)

Multiline strings:

Strings can span multiple lines. The content includes all characters between the quotes:

```
let msg = "This is
    a multi-line
        string"
```

String interpolation:

Expressions enclosed in curly brackets {} inside a string are evaluated and their results are inserted into the string:

```
let name = "world"
let greeting = "Hello, {name}!"          // "Hello, world!"
let result = "2 + 2 = {2 + 2}"          // "2 + 2 = 4"
```

To include literal curly brackets in a string, escape them with a backslash:

```
print("Use \{braces\} for interpolation")   // prints: Use {braces} for interpolation
```

Format specifiers:

String interpolation supports optional format specifiers after a colon:

```
let pi = 3.14159
print("{pi:5.2f}")          // formatted output
```

(see *String Builder* for more details on string interpolation).

2.2.6 Comments

A comment is text that the compiler ignores but is useful for programmers.

Block comments start with /* and end with */. Block comments can span multiple lines and can be nested:

```
/*
This is a multiline comment.
/* This is a nested comment. */
```

(continues on next page)

(continued from previous page)

```
These lines are all ignored by the compiler.
*/
```

Line comments start with // and extend to the end of the line:

```
// This is a single-line comment.
var x = 42  // This is also a comment.
```

2.2.7 Automatic Semicolons

Daslang automatically inserts semicolons at the end of lines when the code is enclosed in curly braces, unless the line is also inside parentheses or brackets:

```
def foo {
    var a = 0       // semicolon inserted automatically
    var b = 1;      // explicit semicolon (redundant but valid)
    var c = ( 1     // no automatic semicolon here (inside parentheses)
        + 2 ) * 3   // semicolon inserted after the closing parenthesis
}
```

This means that expressions can be split across multiple lines when parentheses or brackets keep them together:

```
let result = (
    some_long_function_name(arg1, arg2)
    + another_value
)
```

2.3 Values and Data Types

Daslang is a strong, statically typed language. All variables have a type. Daslang's basic POD (plain old data) data types are:

```
int, uint, float, bool, double, int64, uint64
int2, int3, int4, uint2, uint3, uint4, float2, float3, float4,
range, urange, range64, urange64
```

All PODs are represented with machine register/word. All PODs are passed to functions by value.

Daslang's storage types are:

```
int8, uint8, int16, uint16 - 8/16-bits signed and unsigned integers
```

They can't be manipulated, but can be used as storage type within structs, classes, etc.

Daslang's other types are:

```
string, das_string, struct, pointers, references, block, lambda, function pointer,
array, table, tuple, variant, iterator, bitfield
```

All Daslang's types are initialized with zeroed memory by default.

2.3.1 Integer

An integer represents a 32-bit (un)signed number:

```
let a = 123    // decimal, integer
let u = 123u   // decimal, unsigned integer
let h = 0x0012 // hexadecimal, unsigned integer
let o = 075    // octal, unsigned integer

let a = int2(123, 124)    // two integers type
let u = uint2(123u, 124u) // two unsigned integer type
```

2.3.2 Float

A float represents a 32-bit floating point number:

```
let a = 1.0
let b = 0.234
let a = float2(1.0, 2.0)
```

2.3.3 Bool

A bool is a double-valued (Boolean) data type. Its literals are `true` and `false`. A bool value expresses the validity of a condition (tells whether the condition is true or false):

```
let a = true
let b = false
```

All conditionals (if, elif, while) work only with the bool type.

2.3.4 String

Strings are an immutable sequence of characters. In order to modify a string, it is necessary to create a new one.

Daslang's strings are similar to strings in C or C++. They are delimited by quotation marks(") and can contain escape sequences (\t, \a, \b, \n, \r, \v, \f, \\, \", \', \0, \x<hh>, \u<hhhh> and \U<hhhhhhhh>):

```
let a = "I'm a string\n"
let a = "I'm also
    a multi-line
        string\n"
```

Strings type can be thought of as a 'pointer to the actual string', like a 'const char *' in C. As such, they will be passed to functions by value (but this value is just a reference to the immutable string in memory).

`das_string` is a mutable string, whose content can be changed. It is simply a builtin handled type, i.e., a std::string bound to Daslang. As such, it passed as reference.

2.3.5 Type Conversion and Casting

Daslang is a strongly typed language with **no implicit type conversions**. All numeric operations require operands of the same type — for example, `int + float` is a compilation error. You must convert explicitly:

```
let i = 42
let f = float(i) + 1.0     // explicit int -> float
let i2 = i + int(1.0)      // explicit float -> int
```

Explicit numeric casts

Any numeric type can be explicitly converted to any other numeric type using the target type name as a function:

```
float(42)           // int -> float              (42.0)
int(3.7)            // float -> int, truncates   (3)
double(3.14)        // float -> double
float(3.141f)       // double -> float
uint(42)            // int -> uint
int64(42)           // int -> int64
uint64(42)          // int -> uint64
int8(42)            // int -> int8 (storage type)
uint8(42)           // int -> uint8 (storage type)
int16(42)           // int -> int16 (storage type)
uint16(42)          // int -> uint16 (storage type)
```

Float-to-integer conversion truncates toward zero (like C).

Enumeration casts

Enumerations can be converted to their underlying integer type:

```
enum Color {
    red
    green
    blue
}

let c = Color.green
let i = int(c)              // 1
```

Converting an integer back to an enumeration requires `unsafe` and `reinterpret`:

```
unsafe {
    let c2 = reinterpret<Color>(1)  // Color.green
}
```

String conversion

Any type can be converted to a string via the `string` function:

```
let s = string(42)          // "42"
let s2 = string(3.14)       // "3.14"
```

String interpolation (`{expr}` inside string literals) also converts expressions to text automatically.

To parse strings into numbers, use the functions from `require strings`:

```
require strings
let i = to_int("123")       // 123
let f = to_float("3.14")    // 3.14
```

There is no `int(string)` — use `to_int` instead.

What is NOT allowed

- **No implicit numeric promotion:** `int + float` is a compile error
- **No bool(int):** use a comparison like `x != 0` instead
- **No implicit int-to-float assignment:** `var f :  float = 42` is a compile error; use `float(42)`
- **No int(string):** use `to_int` from the `strings` module

2.3.6 Table

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string; int>
tab["10"] = 10
tab["20"] = 20
tab["some"] = 10
tab["some"] = 20 // replaces the value for 'some' key
```

(see *Tables*).

2.3.7 Array

Arrays are simple sequences of objects. There are static arrays (fixed size) and dynamic arrays (container, size is dynamic). The index always starts from 0:

```
var a = fixed_array(1, 2, 3, 4) // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>            // empty dynamic array
push(b,"some")                  // now it is 1 element of "some"
```

(see *Arrays*).

2.3.8 Struct

Structs are records of data of other types (including structs), similar to C. All structs (as well as other non-POD types, except strings) are passed by reference.

(see *Structs*).

2.3.9 Classes

Classes are similar to structures, but they additionally allow built-in methods and rtti.

(see *Classes*).

2.3.10 Variant

Variant is a special anonymous data type similar to a struct, however only one field exists at a time. It is possible to query or assign to a variant type, as well as the active field value.

(see *Variants*).

2.3.11 Tuple

Tuples are anonymous records of data of other types (including structs), similar to a C++ std::tuple. All tuples (as well as other non-POD types, except strings) are passed by reference.

(see *Tuples*).

2.3.12 Enumeration

An enumeration binds a specific integer value to a name, similar to C++ enum classes.

(see *Enumerations*).

2.3.13 Bitfield

Bitfields are an anonymous data type, similar to enumerations. Each field explicitly represents one bit, and the storage type is always a uint. Queries on individual bits are available on variants, as well as binary logical operations.

(see *Bitfields*).

2.3.14 Function

Functions are similar to those in most other languages:

```
def twice(a: int): int {
    return a + a
}
```

However, there are generic (templated) functions, which will be 'instantiated' during function calls by type inference:

```
def twice(a) {
    return a + a
}

let f = twice(1.0) // 2.0 float
let i = twice(1)   // 2 int
```

(see *Functions*).

2.3.15 Reference

References are types that 'reference' (point to) some other data:

```
def twice(var a: int&) {
    a = a + a
}
var a = 1
twice(a) // a value is now 2
```

All structs are always passed to functions arguments as references.

2.3.16 Pointers

Pointers are types that 'reference' (point to) some other data, but can be null (point to nothing) (see *Pointers*). In order to work with actual value, one need to dereference it using the dereference or safe navigation operators. Dereferencing will panic if a null pointer is passed to it. Pointers can be created using the new operator, or with the C++ environment.

```
def twice(var a: int&) {
    a = a + a
}
def twicePointer(var a: int?) {
    twice(*a)
}

struct Foo {
    x: int
}

def getX(foo: Foo?) { // it returns either foo.x or -1, if foo is null
   return foo?.x ?? -1
}
```

2.3.17 Smart Pointers

Smart pointers (`smart_ptr<T>`) are reference-counted pointers to C++-managed (handled) types. They are **not** available for regular Daslang structs or classes — only for types registered as handled types from the C++ side (such as AST node types in `daslib/ast`).

Smart pointers are primarily used in the macro and AST manipulation context:

```
require ast

var inscope expr : smart_ptr<ExprConstInt> <- new ExprConstInt(value=42)
```

The key properties of smart pointers:

- They maintain a reference count and automatically release the object when the count reaches zero
- They can be moved but not copied via `<-`
- Dereferencing works the same as regular pointers (`*ptr` and `ptr.field`)
- Moving from a smart pointer value requires `unsafe` unless the value is a `new` expression

Because `strict_smart_pointers` is enabled by default, smart pointer variables must be declared with `inscope` to ensure automatic cleanup:

```
var inscope a <- new ExprConstInt(value=1)   // create - safe, no unsafe needed
var inscope b <- a                           // move - safe, a becomes null
unsafe {
    var inscope c <- some_function()         // move from function result - unsafe
}
```

Ownership transfer functions

Daslang provides built-in functions for safe smart pointer ownership transfer. These avoid the need for `unsafe` blocks when reassigning smart pointers that already hold a value:

move(dest, src)
: Transfers ownership from `src` into `dest`. If `dest` already holds a value, its reference count is decremented. After the call, `src` becomes null. Both arguments must be existing smart pointer variables:

```
var inscope a <- new ExprConstInt(value=1)
var inscope b <- new ExprConstInt(value=2)
b |> move <| a       // b now holds what a held; old b is released; a is null
```

move_new(dest, src)
: Transfers ownership from a newly created smart pointer into `dest`. If `dest` already holds a value, its reference count is decremented. This is the idiomatic way to replace the contents of a smart pointer field or variable:

```
var inscope fn <- find_function("foo")
fn |> move_new <| new Function(name := "bar")   // fn now holds the new Function
```

It can also be called in function-call style:

```
move_new(fn) <| new Function(name := "bar")
```

smart_ptr_clone(dest, src)
: Clones (increments the reference count of) `src` into `dest`. Both `dest` and `src` remain valid after the call. If `dest` already held a value, it is released.

`smart_ptr_use_count(ptr)`
: Returns the current reference count of the smart pointer as a `uint`.

Smart pointer types frequently appear in `daslib/ast` and `daslib/ast_boost` when building or transforming AST nodes in macros.

2.3.18 Iterators

Iterators are a sequence which can be traversed, and associated data retrieved. They share some similarities with C++ iterators.

(see *Iterators*).

See also:

Structs, *Tuples*, and *Variants* for composite types, *Arrays* and *Tables* for container types, *Aliases* for type alias declarations, *Bitfields* for the bitfield type.

2.4 Pointers

Daslang provides nullable pointer types for heap-allocated data, optional references, and low-level memory access. Pointer operations split into two categories: **safe** operations that work without `unsafe`, and **unsafe** operations that require an `unsafe` block.

2.4.1 Pointer types

Type	Description
`T?`	Nullable pointer to type `T`
`T? const`	Const pointer — cannot modify the pointed-to value
`T?#`	Temporary pointer (from `safe_addr`) — cannot escape scope
`void?`	Untyped pointer — must `reinterpret` to use

Pointer types are declared by appending `?` to any type:

```
var p : int?              // pointer to int - null by default
var ps : Point?           // pointer to struct
var vp : void?            // void pointer
```

All pointers default to `null` when uninitialized.

2.4.2 Creating pointers

new

`new` allocates on the heap and returns `T?`:

```
var p = new Point(x = 3.0, y = 4.0)   // p is Point?
var q = new Point()                    // default field values
```

Heap pointers must be released with `delete` (see *Deletion*) or declared with `var inscope` for automatic cleanup:

```
var inscope pt = new Point(x = 1.0, y = 2.0)
// pt is automatically deleted at scope exit
```

addr

addr(x) returns a pointer to an existing variable. **Requires unsafe.**

```
var x = 42
unsafe {
    var p = addr(x)   // p is int?
    *p = 100           // modifies x
}
```

The pointer is valid only while the variable is alive — using it after the variable goes out of scope is undefined behavior.

safe_addr

safe_addr from daslib/safe_addr returns a temporary pointer (T?#) without requiring unsafe. The compiler validates that the argument is a local or global variable (not a field of a temporary):

```
require daslib/safe_addr
var a = 13
var p = safe_addr(a)   // p is int?# (temporary pointer)
print("{*p}\n")
```

Temporary pointers cannot be stored in containers or returned from functions.

2.4.3 Dereferencing

*p or deref(p) follows the pointer to the value. Both panic if the pointer is null:

```
*p         // dereference
deref(p)   // same thing
```

For struct pointers, . auto-dereferences — no -> operator is needed:

```
var inscope pt = new Point(x = 5.0, y = 6.0)
print("{pt.x}\n")     // 5 - same as (*pt).x
pt.x = 10.0           // modify through auto-deref
```

2.4.4 Null safety

Null checks

Pointers can be compared to null:

```
if (p != null) {
    print("{*p}\n")      // safe - we checked
}
```

Null dereference panics at runtime and can be caught with `try`/`recover`:

```
try {
    var np : int?
    print("{*np}\n")
} recover {
    print("caught null dereference\n")
}
```

Safe navigation `?.`

`?.` returns `null` instead of panicking when the pointer is null:

```
p?.x          // returns x if p is non-null, null otherwise
a?.b?.c       // chains - short-circuits on first null
```

Safe navigation results are themselves nullable, so combine with `??` for a concrete fallback:

```
let val = p?.x ?? -1      // -1 if p is null
```

Null coalescing `??`

`??` provides a default value when the left side is null:

```
let x = p ?? default_value
```

For pointer dereference:

```
let x = *p ?? 0     // 0 if p is null
```

2.4.5 Deletion

`delete` frees heap memory and sets the pointer to null. **Requires unsafe.**

```
var p = new Point()
unsafe {
    delete p        // frees memory, p becomes null
}
```

Prefer `var inscope` for automatic cleanup — it adds a `finally` block that deletes the pointer when the scope exits:

```
var inscope p = new Point()
// p is automatically deleted at end of scope
```

2.4.6 Pointer arithmetic

All pointer arithmetic **requires unsafe**. No bounds checking is performed.

Indexing

`p[i]` accesses the `i`-th element at the pointer's address:

```
var arr <- [10, 20, 30, 40, 50]
unsafe {
    var p = addr(arr[0])
    print("{p[0]}, {p[2]}\n")     // 10, 30
}
```

Increment and addition

```
unsafe {
    ++ p          // advance pointer by one element
    p += 3        // advance by three elements
}
```

> **Warning:** Pointer arithmetic can easily cause out-of-bounds access or invalid pointer states. Use array bounds-checked access whenever possible.

2.4.7 Void pointers

`void?` is an untyped pointer — equivalent to `void*` in C/C++. It is used for opaque handles and C/C++ interop. You must `reinterpret` it back to a typed pointer before dereferencing:

```
unsafe {
    var x = 123
    var px = addr(x)
    var vp : void? = reinterpret<void?> px     // erase type
    var px2 = reinterpret<int?> vp             // restore type
    print("{*px2}\n")                          // 123
}
```

2.4.8 intptr

`intptr(p)` converts any pointer (raw or smart) to a `uint64` integer representing its memory address:

```
let address = intptr(p)   // uint64
```

Useful for debugging, logging, pointer identity comparisons, or hashing.

2.4.9 reinterpret

`reinterpret<T>` performs a raw bit cast between types of the same size. **Requires unsafe.** It does not convert values — it reinterprets the raw bits:

```
unsafe {
    let f = 1.0
    let bits = reinterpret<int> f      // IEEE 754: 0x3f800000
    let back = reinterpret<float> bits // 1.0
}
```

Can also cast between pointer types:

```
unsafe {
    var p : int? = addr(x)
    var vp = reinterpret<void?> p    // to void?
    var p2 = reinterpret<int?> vp    // back to int?
}
```

2.4.10 Type info

Several `typeinfo` queries test pointer properties at compile time:

```
typeinfo(is_pointer p)       // true if p is a pointer type
typeinfo(is_smart_ptr p)     // true if p is a smart_ptr<T>
typeinfo(is_void_pointer p)  // true if p is void?
typeinfo(can_delete_ptr p)   // true if delete is valid for p
```

2.4.11 Summary

Safe (no unsafe required):

- `new T()` — heap allocate, returns `T?`
- `*p` / `deref(p)` — dereference (panics if null)
- `p.field` — auto-deref field access
- `p?.field` — safe navigation (null-propagating)
- `p ?? default` — null coalescing
- `safe_addr(x)` — temporary pointer (`T?#`)
- `var inscope p = new T()` — automatic cleanup
- `intptr(p)` — pointer to integer

Unsafe (requires unsafe block):

- `addr(x)` — address of variable
- `delete p` — free heap memory
- `p[i]` — pointer indexing
- `++ p` / `p += N` — pointer arithmetic
- `reinterpret<T>` — raw bit cast

See also:

Unsafe for the full list of unsafe operations.

Values and Data Types for smart pointers (`smart_ptr<T>`).

Temporary types for temporary pointers (`T?#`) and `safe_addr`.

Tutorial: Pointers for a hands-on walkthrough.

2.5 Constants, Enumerations, Global variables

Daslang allows you to bind constant values to a global variable identifier. Whenever possible, all constant global variables will be evaluated at compile time. There are also enumerations, which are strongly typed constant collections similar to enum classes in C++.

2.5.1 Constant

Constants bind a specific value to an identifier. Constants are global variables whose values cannot be changed.

Constants are declared with the following syntax:

```
let foobar = 100
let floatbar = 1.2
let stringbar = "I'm a constant string"
let blah = "I'm string constant which is declared on the same line as variable"
```

Constants are always globally scoped from the moment they are declared. Any subsequent code can reference them.

You cannot change the value of a constant.

Constants can be `shared`:

```
let shared blah <- ["blah","blahh","blahh"]
```

Shared constants point to the same memory across all `Context` instances and are initialized once, during the first context initialization.

2.5.2 Global variable

Mutable global variables are defined as:

```
var foobar = 100
var barfoo = 100
```

Their usage can be switched on and off on a per-project basis via `CodeOfPolicies`.

Local static variables can be declared via the `static_let` macro:

```
require daslib/static_let

def foo {
    static_let {
        var bar = 13
```

(continues on next page)

(continued from previous page)

```
    }
    bar = 14
}
```

Variable bar in the example above is effectively a global variable. However, it's only visible inside the scope of the corresponding static_let macro.

Global variables can be private or public

```
let public foobar = 100
let private barfoo = 100
```

If not specified, global variables inherit module publicity (i.e. in public modules global variables are public, and in private modules global variables are private).

2.5.3 Enumeration

An enumeration binds a specific value to a name. Enumerations are also evaluated at compile time and their value cannot be changed.

An enum declaration introduces a new enumeration to the program. Enumeration values can only be compile time constant expressions. It is not required to assign specific value to enum:

```
enum Numbers {
    zero     // will be 0
    one      // will be 1
    two      // will be 2
    ten = 9+1 // will be 10, since its explicitly specified
}
```

Enumerations can be private or public:

```
enum private Foo {
    fooA
    fooB
}

enum public Bar {
    barA
    barB
}
```

If not specified, enumerations inherit module publicity (i.e. in public modules enumerations are public, and in private modules enumerations are private).

An enum name itself is a strong type, and all enum values are of this type. An enum value can be addressed as 'enum name' followed by exact enumeration

```
let one: Numbers = Numbers.one
```

An enum value can be converted to an integer type with an explicit cast

```
let one: Numbers = Numbers.one
assert(int(one) == 1)
```

Enumerations can specify one of the following storage types: `int`, `int8`, `int16`, `uint`, `uint8`, or `uint16`:

```
enum Characters : uint8 {
    ch_a = 'A'
    ch_b = 'B'
}
```

Enumeration values will be truncated down to the storage type.

The `each_enum` iterator iterates over specific enumeration type values. Any enum element needs to be provided to specify the enumeration type:

```
for ( x in each_enum(Characters.ch_a) ) {
    print("x = {x}\n")
}
```

See also:

Datatypes for a list of built-in types including enum storage types, *Iterators* for `each_enum` and other iteration patterns, *Pattern matching* for matching on enumeration values, *Contexts* for shared constant initialization across contexts, *Program structure* for the overall layout of global declarations.

2.6 Statements

A Daslang program is a sequence of statements. Statements in Daslang are comparable to those in C-family languages (C/C++, Java, C#, etc.): assignments, function calls, control flow, and declarations. There are also Daslang-specific statements such as `with`, `assume`, `static_if`, and generators.

Statements can be separated with a newline or a semicolon `;`.

2.6.1 Visibility Block

A sequence of statements delimited by curly brackets is called a *visibility block*. Variables declared inside a visibility block are only visible within that block:

```
def foo {
    var x = 1           // x is visible here
    {
        var y = 2       // y is visible here
        x += y
    }
    // y is no longer visible
}
```

2.6.2 Control Flow Statements

if/elif/else

Conditionally execute a block depending on the result of a boolean expression:

```
if ( a > b ) {
    a = b
} elif ( a < b ) {
    b = a
} else {
    print("equal\n")
}
```

Daslang has a strong boolean type. Only expressions of type `bool` can be used as conditions.

One-liner if syntax:

A single-expression if statement can be written on one line:

```
if ( a > b ) { a = b }
```

Postfix if syntax:

The condition can be written after the statement:

```
a = b if ( a < b )
```

Ternary-style if:

A full ternary expression can use if/else inline:

```
return 13 if ( a == 42 ) else return 7
```

static_if / static_elif

`static_if` evaluates its condition at compile time. It is used in generic functions to select code paths based on type properties. Branches that do not match are removed from the compiled output, and do not need to be valid for the given types:

```
def describe(a) {
    static_if ( typeinfo(is_pointer type<a>) ) {
        print("pointer\n")
    } static_elif ( typeinfo(is_ref_type type<a>) ) {
        print("reference type\n")
    } else {
        print("value type\n")
    }
}
```

Unlike regular `if`, `static_if` does not require its condition to be of type `bool` — it only requires a compile-time constant expression.

`static_if` is the primary mechanism for conditional compilation in generic code (see *Generic Programming*).

while

Execute a block repeatedly while a boolean condition is true:

```
var i = 0
while ( i < 10 ) {
    print("{i}\n")
    i++
}

while ( true ) {
    if ( done() ) {
        break
    }
}
```

The condition must be of type `bool`.

2.6.3 Ranged Loops

for

Execute a block once for each element of one or more iterable sources:

```
for ( i in range(0, 10) ) {
    print("{i}\n")          // prints 0 through 9
}
```

Multiple iterables can be traversed in parallel. Iteration stops when the shortest source is exhausted:

```
var a : array<int>
var b : int[10]
resize(a, 4)
for ( l, r in a, b ) {
    print("{l} == {r}\n")   // iterates over 4 elements (length of a)
}
```

Table keys and values can be iterated using the `keys` and `values` functions:

```
var tab <- { "one"=>1, "two"=>2 }
for ( k, v in keys(tab), values(tab) ) {
    print("{k}: {v}\n")
}
```

Iterable types include ranges, arrays, fixed arrays, tables (via `keys`/`values`), strings (via `each`), enumerations (via `each_enum`), and custom iterators (see *Iterators*).

Any type can be made directly iterable by defining an `each` function that accepts the type and returns an `iterator`. When such a function exists, `for (x in y)` is equivalent to `for (x in each(y))`:

```
struct Foo {
    data : array<int>
}
```

(continues on next page)

(continued from previous page)

```
def each ( f : Foo ) : iterator<int&> {
    return each(f.data)
}

var f = Foo(data <- [1, 2, 3])
for ( x in f ) {            // calls each(f) automatically
    print("{x}\n")
}
```

Tuple expansion in for loops:

When iterating over containers of tuples, elements can be unpacked directly:

```
var data <- [(1, 2.0, "three"), (4, 5.0, "six")]
for ( (a, b, c) in data ) {
    print("{a} {b} {c}\n")
}
```

2.6.4 break

Terminate the enclosing `for` or `while` loop immediately:

```
for ( x in arr ) {
    if ( x == target ) {
        break
    }
}
```

`break` cannot cross block boundaries. Using `break` inside a block passed to a function is a compilation error.

2.6.5 continue

Skip the rest of the current loop iteration and proceed to the next one:

```
for ( x in range(0, 10) ) {
    if ( x % 2 == 0 ) {
        continue
    }
    print("{x}\n")      // prints only odd numbers
}
```

2.6.6 return

Terminate the current function, block, or lambda and optionally return a value:

```
def add(a, b : int) : int {
    return a + b
}
```

All return statements in a function must return the same type. If no expression is given, the function is assumed to return `void`:

```
def greet(name : string) {
    print("Hello, {name}!\n")
    return
}
```

Move-on-return transfers ownership of a value using the `<-` operator:

```
def make_array : array<int> {
    var result : array<int>
    result |> push(1)
    result |> push(2)
    return <- result
}
```

In generator blocks, `return` must always return a boolean expression where `false` indicates the end of generation (see *Generators*).

2.6.7 yield

Output a value from a generator and suspend its execution until the next iteration. `yield` can only be used inside `generator` blocks:

```
var gen <- generator<int>() <| $ {
    yield 0
    yield 1
    return false    // end of generation
}
```

Move semantics are also supported:

```
yield <- some_array
```

(see *Generators*).

2.6.8 pass

An explicit no-operation statement. `pass` does nothing and can be used as a placeholder in blocks that are intentionally empty:

```
def todo_later {
    pass
}
```

2.6.9 finally

Declare a block of code that executes when the enclosing scope exits, regardless of how it exits (normal flow, `break`, `continue`, or `return`):

```
def test(a : array<int>; target : int) : int {
    for ( x in a ) {
        if ( x == target ) {
            return x
        }
    }
    return -1
} finally {
    print("search complete\n")
}
```

`finally` can be attached to any block, including loops:

```
for ( x in data ) {
    if ( x < 0 ) {
        break
    }
} finally {
    print("loop done\n")
}
```

A `finally` block cannot contain `break`, `continue`, or `return` statements.

The `defer` macro from `daslib/defer` provides a convenient way to add cleanup code to the current scope's `finally` block:

```
require daslib/defer

def process {
    var resource <- acquire()
    defer() {
        release(resource)
    }
    // ... use resource ...
}   // release(resource) is called here
```

Multiple `defer` statements execute in reverse order (last-in, first-out).

The `defer_delete` macro adds a `delete` statement for its argument without requiring a block.

2.6.10 Local Variable Declarations

Local variables can be declared at any point inside a function. They exist from the point of declaration until the end of their enclosing visibility block.

`let` declares a read-only (constant) variable, and `var` declares a mutable variable:

```
let pi = 3.14159        // constant, cannot be modified
var counter = 0         // mutable, can be modified
counter++
```

Variables can be initialized with copy (=), move (<-), or clone (:=) semantics:

```
var a <- [1, 2, 3]      // move: a now owns the array
var b : array<int>
b := a                  // clone: b is a deep copy of a
```

If a type is specified, the variable is typed explicitly. Otherwise, the type is inferred from the initializer:

```
var x : int = 42        // explicit type
var y = 42              // inferred as int
var z : float           // explicit type, initialized to 0.0
```

inscope variables:

When `inscope` is specified, a `delete` statement is automatically added to the `finally` section of the enclosing block:

```
var inscope resource <- acquire()
// ... use resource ...
// delete resource is called automatically at end of block
```

`inscope` cannot appear directly in a loop block, since the `finally` section of a loop executes only once.

Variable name aliases (aka):

The `aka` keyword creates an alternative name (alias) for a variable. Both names refer to the same value. This works in `let`, `var`, and `for` declarations:

```
var a aka alpha = 42
print("{alpha}\n")          // prints 42 - alpha is the same variable as a

for (x aka element in [1,2,3]) {
    print("{element}\n")    // element is the same as x
}
```

Tuple expansion:

Variables can be unpacked from tuples:

```
var (x, y, z) = (1, 2.0, "three")
// x is int, y is float, z is string
```

2.6.11 assume

The `assume` statement creates a named alias for an expression without creating a new variable. Every use of the alias substitutes the original expression:

```
var data : array<array<int>>
assume inner = data[0]
inner |> push(42)       // equivalent to data[0] |> push(42)
```

`assume` is particularly useful for simplifying repeated access to nested data:

```
assume cfg = settings.graphics.resolution
print("width={cfg.width}, height={cfg.height}\n")
```

Note: `assume` does not create a variable — it creates a textual substitution. The expression is re-evaluated at each point of use.

2.6.12 with

The `with` statement brings the fields of a structure, class, or handled type into the current scope, allowing them to be accessed without a prefix:

```
struct Player {
    x, y : float
    health : int
}

def reset(var p : Player) {
    with ( p ) {
        x = 0.0
        y = 0.0
        health = 100
    }
}
```

Without `with`, the same code would require the `p.` prefix on each field access.

Multiple `with` blocks can be nested. If field names conflict, the innermost `with` takes precedence.

2.6.13 delete

The `delete` statement invokes the finalizer for a value, releasing any resources it holds. After deletion, the value is zeroed:

```
var arr <- [1, 2, 3]
delete arr              // arr is now empty
```

Deleting pointers is an unsafe operation because other references to the same data may still exist:

```
var p = new Foo()
unsafe {
    delete p            // p is set to null, memory is freed
}
```

(see *Finalizers*).

2.6.14 Function Declaration

Functions are declared with the `def` keyword:

```
def add(a, b : int) : int {
    return a + b
}

def greet {
    print("hello\n")
}
```

(see *Functions* for a complete description of function features).

2.6.15 try/recover

Enclose a block of code that may trigger a runtime panic, such as null pointer dereference or out-of-bounds array access:

```
try {
    var p : Foo?
    print("{*p}\n")         // would panic: null pointer dereference
} recover {
    print("recovered from panic\n")
}
```

> **Warning:** `try/recover` is **not** a general error-handling mechanism and should not be used for control flow. It is designed for catching runtime panics (similar to Go's `recover`). In production code, these situations should be prevented rather than caught.

2.6.16 panic

Trigger a runtime panic with an optional message:

```
panic("something went very wrong")
```

The panic message is available in the runtime log. A panic can be caught by a `try/recover` block.

2.6.17 label and goto

Daslang supports numeric labels and goto for low-level control flow:

```
label 0:
    print("start\n")
    goto label 1
label 1:
    print("end\n")
```

Labels use integer identifiers. Computed goto is also supported:

```
goto label_expression
```

Warning: Labels and goto are low-level constructs primarily used in generated code (such as generators). They are generally not recommended for regular application code.

2.6.18 Expression Statement

Any expression is also valid as a statement. The result of the expression is discarded:

```
foo()               // function call as statement
a + b               // valid but result is unused
arr |> push(42)     // pipe expression as statement
```

2.6.19 Global Variables

Global variables are declared at module scope with `let` (constant) or `var` (mutable):

```
let MAX_SIZE = 1024
var counter = 0
```

Global variables are initialized once during script initialization (or each time `init` is manually called on the context).

`shared` indicates that the variable's memory is shared between multiple Context instances and initialized only once:

```
let shared lookup_table <- generate_table()
```

`private` indicates the variable is not visible outside its module:

```
var private internal_state = 0
```

(see *Constants & Enumerations* for more details).

2.6.20 enum

Declare an enumeration — a set of named integer constants:

```
enum Color {
    Red
    Green
    Blue
}
```

(see *Constants & Enumerations*).

2.6.21 typedef

Declare a type alias:

```
typedef Vec3 = float3
typedef IntPair = tuple<int; int>
```

Type aliases can also be declared locally inside functions or structure bodies (see *Type Aliases*).

See also:

Expressions for expression syntax and operators, *Functions* for function declarations using `def`, *Iterators* for `for` loop iteration patterns, *Generators* for `yield` in generator functions.

2.7 Expressions

2.7.1 Assignment

Daslang provides three kinds of assignment:

Copy assignment (=) performs a bitwise copy of the value:

```
a = 10
```

Copy assignment is only available for POD types and types that support copying. Arrays, tables, and other container types cannot be copied — use move or clone instead.

Move assignment (<-) transfers ownership of a value, zeroing the source:

```
var b = new Foo()
var a : Foo?
a <- b              // a now points to the Foo instance, b is null
```

Move is the primary mechanism for transferring ownership of heavy types such as arrays and tables. Some handled types may be movable but not copyable.

Clone assignment (:=) creates a deep copy of the value:

```
var a : array<int>
a := b              // a is now a deep copy of b
```

Clone is syntactic sugar for calling the `clone` function. For POD types, clone falls back to a regular copy.

(see *Move, Copy, and Clone* for a complete guide, and *Clone* for detailed cloning rules).

2.7.2 Operators

Arithmetic

Daslang supports the standard arithmetic operators +, -, *, /, and % (modulo). Compound assignment operators +=, -=, *=, /=, %= and increment/decrement operators ++ and -- are also available:

```
a += 2          // equivalent to a = a + 2
x++             // equivalent to x = x + 1
--y             // prefix decrement
```

All arithmetic operators are defined for numeric and vector types (`int`, `uint`, `int2`–`int4`, `uint2`–`uint4`, `float`–`float4`, `double`).

Relational

Relational operators compare two values and return a `bool` result: ==, !=, <, <=, >, >=:

```
if ( a == b ) { print("equal\n") }
if ( x < 0 ) { print("negative\n") }
```

Logical

Logical operators work with `bool` values:

- && — logical AND. Returns `false` if the left operand is `false`; otherwise evaluates and returns the right operand.
- || — logical OR. Returns `true` if the left operand is `true`; otherwise evaluates and returns the right operand.
- ^^ — logical XOR. Returns `true` if the operands differ.
- ! — logical NOT. Returns `false` if the value is `true`, and vice versa.

Compound assignment forms are available: &&=, ||=, ^^=.

Important: && and || use short-circuit evaluation — the right operand is not evaluated if the result can be determined from the left operand alone. Unlike their C++ counterparts, &&= and ||= also short-circuit the right side.

Bitwise Operators

Daslang supports C-like bitwise operators for integer types:

- & — bitwise AND
- | — bitwise OR
- ^ — bitwise XOR
- ~ — bitwise NOT (complement)
- << — shift left

- >> — shift right
- <<< — rotate left
- >>> — rotate right

Compound assignment forms: &=, |=, ^=, <<=, >>=, <<<=, >>>=:

```
let flags = 0xFF & 0x0F     // 0x0F
let rotated = value <<< 3   // rotate left by 3 bits
```

Pipe Operators

Pipe operators pass a value as the first (right pipe) or last (left pipe) argument to a function call:

- |> — right pipe. `x |> f(y)` is equivalent to `f(x, y)`
- <| — left pipe. `f(y) <| x` is equivalent to `f(y, x)`

```
def addX(a, b) {
    return a + b
}

let t = 12 |> addX(2) |> addX(3)   // addX(addX(12, 2), 3) = 17
```

Left pipe is commonly used to pass blocks and lambdas to functions:

```
def doSomething(blk : block) {
    invoke(blk)
}

doSomething() <| $ {
    print("hello\n")
}
```

In gen2 syntax a block or lambda that immediately follows a function call is automatically piped as the last argument, so the explicit <| can be omitted. Parameterless blocks also do not need the $ prefix:

```
doSomething() {                     // same as doSomething() <| $ { ... }
    print("hello\n")
}

build_string() $(var writer) {      // block with parameters - $ is still required
    write(writer, "hello")
}

sort(arr) @(a, b) => a < b          // lambda - @ is still required
```

This shorthand works with lambdas (@) and no-capture lambdas (@@) as well. The explicit <| is still needed when a block is passed to an expression (e.g. `<| new ...`) or for generator bodies (`generator<T>() <| $() { ... }`).

The `lpipe` macro from `daslib/lpipe` allows piping to the expression on the previous line:

```
require daslib/lpipe

def main {
```

(continues on next page)

(continued from previous page)

```
    print()
    lpipe() <| "this is a string"
}
```

Interval Operator

The `..` operator creates a range from two values:

```
let r = 1 .. 10     // equivalent to interval(1, 10)
```

By default, `interval(a, b : int)` returns `range(a, b)` and `interval(a, b : uint)` returns `urange(a, b)`. Custom `interval` functions or generics can be defined for other types.

Null-Coalescing Operator (??)

The `??` operator returns the dereferenced value of the left operand if it is not `null`, otherwise returns the right operand:

```
var p : int?
let value = p ?? 42     // value is 42 because p is null
```

This is equivalent to:

```
let value = (p != null) ? *p : 42
```

`??` evaluates expressions left to right until the first non-null value is found (similar to how `||` works for booleans).

The `??` operator has lower precedence than `|`, following C# convention.

Ternary Operator (? :)

The ternary operator conditionally evaluates one of two expressions:

```
let result = (a > b) ? a : b    // returns the larger value
```

Only the selected branch is evaluated.

Null-Safe Navigation (?. and ?[)

The `?.` operator accesses a field of a pointer only if the pointer is not null. If the pointer is null, the result is null:

```
struct Foo {
    x : int
}

struct Bar {
    fooPtr : Foo?
}

def getX(bar : Bar?) : int {
    return bar?.fooPtr?.x ?? -1     // returns -1 if bar or fooPtr is null
}
```

The `?[` operator provides null-safe indexing into tables:

```
var tab <- { "one"=>1, "two"=>2 }
let i = tab?["three"] ?? 3      // returns 3 because "three" is not in the table
```

Both operators can be used on the left side of an assignment with `??`:

```
var dummy = 0
bar?.fooPtr?.x ?? dummy = 42    // writes to dummy if navigation fails
```

Type Operators (is, as, ?as)

The `is` operator checks the active variant case:

```
variant Value {
    i : int
    f : float
}
var v = Value(i = 42)
if ( v is i ) { print("it's an int\n") }
```

The `as` operator accesses the value of a variant case. It panics if the wrong case is active:

```
let x = v as i      // returns 42
```

The `?as` operator is a safe version of `as` that returns null if the case does not match:

```
let x = v ?as f ?? 0.0     // returns 0.0 because v is not f
```

These operators can also be used with classes and the `is`/`as` operator overloading mechanism (see *Pattern Matching*).

is type<T>

The `is type<T>` expression performs a compile-time type check. It returns `true` if the expression's type matches the specified type, and `false` otherwise:

```
let a = 42
let b = 3.14
print("{a is type<int>}\n")     // true
print("{b is type<float>}\n")   // true
print("{b is type<int>}\n")     // false
```

This is useful in generic functions to branch on the actual type of a parameter:

```
def describe(x) {
    static_if (x is type<int>) {
        print("an integer\n")
    } static_elif (x is type<float>) {
        print("a float\n")
    } else {
        print("something else\n")
    }
}
```

Cast, Upcast, and Reinterpret

cast performs a safe downcast from a parent structure type to a derived type:

```
var derived : Derived = Derived()
var base : Base = cast<Base> derived
```

upcast performs an unsafe upcast from a base type to a derived type. This requires `unsafe` because the actual runtime type may not match:

```
unsafe {
    var d = upcast<Derived> base_ref
}
```

reinterpret reinterprets the raw bits of a value as a different type. This is unsafe and should be used with extreme caution:

```
unsafe {
    let p = reinterpret<void?> 13
}
```

Dereference

The * prefix operator dereferences a pointer, converting it to a reference. Dereferencing a null pointer causes a panic:

```
var p = new Foo()
var ref = *p        // ref is Foo&
```

The `deref` keyword can be used as an alternative:

```
var ref = deref(p)
```

Address-of

The `addr` function takes the address of a value, creating a pointer. This is an unsafe operation:

```
unsafe {
    var x = 42
    var p = addr(x)     // p is int?
}
```

Dot Operator Bypass

Smart pointers (`smart_ptr<T>`) are accessed the same way as regular pointers — using `.` for field access and `?.` for null-safe field access.

The `..` operator bypasses any `.` operator overloading and accesses the underlying field directly. This is useful when a handled type defines a custom `.` operator but you need to reach the actual field:

```
sp..x = 42      // accesses field x directly, skipping any . overload
```

Safe Index (?[)

The `?[` operator provides null-safe indexing. If the pointer or table key is null or missing, the result is `null` instead of a panic:

```
var tab <- { "one"=>1, "two"=>2 }
let i = tab?["three"] ?? 3      // returns 3 because "three" is not in the table
```

Unsafe Expression

Individual expressions can be marked as unsafe without wrapping an entire block:

```
let p = unsafe(addr(x))
```

This is equivalent to wrapping the expression in an `unsafe { }` block.

Operators Precedence

post++ post-- . -> ?. ?[*(deref)	highest
|> <|	
is as	
- + ~ ! ++ --	
??	
/ * %	
+ -	
<< >> <<< >>>	
< <= > >=	
== !=	
&	
^	
|	
&&	
^^	
||	
? :	
+= = -= /= *= %= &= |= ^= <<= >>= <- <<<= >>>= &&= ||= ^^= :=	
.. =>	
,	lowest

2.7.3 Array Initializer

Fixed-size arrays can be created with the `fixed_array` keyword:

```
let a = fixed_array<int>(1, 2)      // int[2]
let b = fixed_array(1, 2, 3)        // inferred as int[3]
```

Dynamic arrays can be created with several syntaxes:

```
let a <- [1, 2, 3]                  // array<int>
let b <- array(1, 2, 3)             // array<int>
let c <- array<int>(1, 2, 3)        // explicitly typed
let d <- [for x in range(0, 10); x * x]  // comprehension
```

(see *Arrays*, *Comprehensions*).

2.7.4 Struct, Class, and Handled Type Initializer

Structures can be initialized by specifying field values:

```
struct Foo {
    x : int = 1
    y : int = 2
}

let a = Foo(x = 13, y = 11)                    // x = 13, y = 11
let b = Foo(x = 13)                            // x = 13, y = 2 (default)
let c = Foo(uninitialized x = 13)               // x = 13, y = 0 (no default init)
```

Arrays of structures can be constructed inline:

```
var arr <- array struct<Foo>((x=11, y=22), (x=33), (y=44))
```

Classes and handled (external) types can also be initialized using this syntax. Classes and handled types cannot use `uninitialized`.

(see *Structs*, *Classes*).

2.7.5 Tuple Initializer

Tuples can be created with several syntaxes:

```
let a = (1, 2.0, "3")                                  // inferred tuple type
let b = tuple(1, 2.0, "3")                              // same as above
let c = tuple<a:int; b:float; c:string>(a=1, b=2.0, c="3")  // named fields
```

(see *Tuples*).

2.7.6 Variant Initializer

Variants are created by specifying exactly one field:

```
variant Foo {
    i : int
    f : float
}

let x = Foo(i = 3)
let y = Foo(f = 4.0)
```

Variants can also be declared as type aliases:

```
typedef Foo = variant<i:int; f:float>
```

(see *Variants*).

2.7.7 Table Initializer

Tables are created by specifying key-value pairs separated by =>:

```
var a <- { 1=>"one", 2=>"two" }
var b <- table("one"=>1, "two"=>2)      // alternative syntax
var c <- table<string; int>("one"=>1)    // explicitly typed
```

All values in a table literal must be of the same type. Similarly, all keys must be of the same type.

(see *Tables*).

2.7.8 default and new

The `default` expression creates a default-initialized value of a given type:

```
var a = default<Foo>            // all fields zeroed, then default initializer called
var b = default<Foo> uninitialized  // all fields zeroed, no initializer
```

The `new` operator allocates a value on the heap and returns a pointer:

```
var p = new Foo()               // Foo? pointer, default initialized
var q = new Foo(x = 13)         // with field initialization
```

`new` can also be combined with array and table literals to allocate them on the heap:

```
var p <- new [1, 2, 3]          // heap-allocated array<int>
```

2.7.9 typeinfo

The `typeinfo` expression provides compile-time type information. It is primarily used in generic functions to inspect argument types:

```
typeinfo(typename type<int>)        // returns "int" at compile time
typeinfo(sizeof type<float3>)       // returns 12
typeinfo(is_pod type<int>)          // returns true
typeinfo(has_field<x> myStruct)     // returns true if myStruct has field x
```

(see *Generic Programming* for a complete list of typeinfo traits).

2.7.10 String Interpolation

Expressions inside curly brackets within a string are evaluated and converted to text:

```
let name = "world"
print("Hello, {name}!")              // Hello, world!
print("1 + 2 = {1 + 2}")            // 1 + 2 = 3
```

Format specifiers can be added after a colon:

```
let pi = 3.14159
print("pi = {pi:5.2f}")             // formatted output
```

To include literal curly brackets, escape them with backslashes:

```
print("Use \{curly\} brackets")     // Use {curly} brackets
```

(see *String Builder*).

See also:

Statements for control flow and variable declarations, *Pattern matching* for `is`/`as`/`?as` operator details, *Datatypes* for a list of types used in expressions.

2.8 Function

Function pointers are first-class values, like integers or strings, and can be stored in table slots, local variables, arrays, and passed as function parameters. Functions themselves are declarations (much like in C++).

2.8.1 Function declaration

Functions are similar to those in most other typed languages:

```
def twice(a: int): int {
    return a+a
}
```

Completely empty functions (without arguments) can be also declared:

```
def foo {
    print("foo")
}

//same as above
def foo() {
    print("foo")
}
```

Daslang can always infer a function's return type. Returning different types is a compilation error:

```
def foo(a:bool) {
    if ( a ) {
        return 1
```

(continues on next page)

(continued from previous page)

```
    } else {
        return 2.0  // error, expecting int
    }
}
```

The return type can be specified explicitly with `:` or `->` — both are equivalent:

```
def add(a, b : int) : int {
    return a + b
}

def add(a, b : int) -> int {   // same as above
    return a + b
}
```

Publicity

Functions can be `private` or `public`

```
def private foo(a:bool)

def public bar(a:float)
```

If not specified, functions inherit module publicity (i.e. in public modules functions are public, and in private modules functions are private).

Function calls

You can call a function by using its name and passing in all its arguments (with the possible omission of the default arguments):

```
def foo(a, b: int) {
    return a + b
}

def bar {
    foo(1, 2) // a = 1, b = 2
}
```

Named Arguments Function call

You can also call a function by using its name and passing all its arguments with explicit names (with the possible omission of the default arguments):

```
def foo(a, b: int) {
    return a + b
}

def bar {
```

(continues on next page)

(continued from previous page)

```
    foo([a = 1, b = 2])  // same as foo(1, 2)
}
```

Named arguments should be still in the same order:

```
def bar {
    foo([b = 1, a = 2])  // error, out of order
}
```

Named argument calls increase the readability of callee code and ensure correctness in refactorings of the existing functions. They also allow default values for arguments other than the last ones:

```
def foo(a:int=13, b: int) {
    return a + b
}

def bar {
    foo([b = 2])  // same as foo(13, 2)
}
```

Function pointer

Pointers to a function use a similar declaration to that of a block or lambda. The type is written as `function` followed by an optional type signature in angle brackets:

```
function < (arg1:int; arg2:float&) : bool >
```

The -> operator can be used instead of : for the return type:

```
function < (arg1:int; arg2:float&) -> bool >   // equivalent
```

If no type signature is specified, `function` alone represents a function pointer with an unspecified signature.

Function pointers can be obtained by using the @@ operator:

```
def twice(a:int) {
    return a + a
}

let fn = @@twice
```

When multiple functions have the same name, a pointer can be obtained by explicitly specifying signature:

```
def twice(a:int) {
    return a + a
}

def twice(a:float) {  // when this one is required
    return a + a
}

let fn = @@<(a:float):float> twice
```

Function pointers can be called via `invoke` or via call notation:

```
let t = invoke(fn, 1)   // t = 2
let t = fn(1)           // t = 2, same as
```

Nameless functions

Pointers to nameless functions can be created with a syntax similar to that of lambdas or blocks (see *Blocks*):

```
let fn <- @@ ( a : int ) {
    return a + a
}
```

Nameless local functions do not capture variables at all:

```
var count = 1
let fn <- @@ ( a : int ) {
    return a + count            // compilation error, can't locate variable count
}
```

Internally, a regular function will be generated:

```
def _localfunction_thismodule_8_8_1`function ( a:int const ) : int {
        return a + a
}

let fn:function<(a:int const):int> const <- @@_localfunction_thismodule_8_8_1`function
```

Generic functions

Generic functions are similar to C++ templated functions. Daslang will instantiate them during the infer pass of compilation:

```
def twice(a) {
    return a + a
}

let f = twice(1.0)  // 2.0 float
let i = twice(1)    // 2 int
```

Generic functions allow code similar to dynamically-typed languages like Python or Lua, while still enjoying the performance and robustness of strong, static typing.

You cannot take the address of a generic function.

Unspecified types can also be written via `auto` notation:

```
def twice(a:auto) {   // same as 'twice' above
    return a + a
}
```

Generic functions can specialize generic type aliases, and use them as part of the declaration:

```
def twice(a:auto(TT)) : TT {
    return a + a
}
```

In the example above, alias TT is used to enforce the return type contract.

Type aliases can be used before the corresponding auto:

```
def summ(base : TT; a:auto(TT)[] ) {
    var s = base
    for ( x in a ) {
        s += x
    }
    return s
}
```

In the example above, TT is inferred from the type of the passed array a, and expected as a first argument base. The return type is inferred from the type of s, which is also TT.

Function overloading

Functions can be specialized if their argument types are different:

```
def twice(a: int) {
    print("int")
    return a + a
}
def twice(a: float) {
    print("float")
    return a + a
}

let i = twice(1)    // prints "int"
let f = twice(1.0)  // prints "float"
```

Declaring functions with the same exact argument list is a compilation-time error.

Functions can be partially specialized:

```
def twice(a:int) {      // int
    return a + a
}
def twice(a:float) {    // float
    return a + a
}
def twice(a:auto[]) {   // any array
    return length(a)*2
}
def twice(a) {          // any other case
    return a + a
}
```

Daslang uses the following rules for matching partially specialized functions:

1. Non-auto is more specialized than auto.

2. If both are non-`auto`, the one without a cast is more specialized.
3. Ones with arrays are more specialized than ones without. If both have an array, the one with the actual value is more specialized than the one without.
4. Ones with a base type of autoalias are less specialized. If both are autoalias, it is assumed that they have the same level of specialization.
5. For pointers and arrays, the subtypes are compared.
6. For tables, tuples and variants, subtypes are compared, and all must be the same or equally specialized.
7. For functions, blocks, or lambdas, subtypes and return types are compared, and all must be the same or equally specialized.

When matching functions, Daslang picks the ones which are most specialized and sorts by substitute distance. Substitute distance is increased by 1 for each argument if a cast is required for the LSP (Liskov substitution principle). At the end, the function with the least distance is picked. If more than one function is left for picking, a compilation error is reported.

Function specialization can be limited by contracts (contract macros):

```
[expect_any_array(blah)]  // array<foo>, [], or dasvector`.... or similar
def print_arr ( blah ) {
    for ( i in range(length(blah)) ) {
        print("{blah[i]}\n")
    }
}
```

In the example above, only arrays will be matched.

It is possible to use boolean logic operations on contracts:

```
[expect_any_tuple(blah) || expect_any_variant(blah)]
def print_blah ...
```

In the example above print_blah will accept any tuple or variant. Available logic operations are `!`, `&&`, `||` and `^^`.

LSP can be explicitly prohibited for a particular function argument via the `explicit` keyword:

```
def foo ( a : Foo explicit ) // will accept Foo, but not any subtype of Foo
```

Default Parameters

Daslang's functions can have default parameters.

A function with default parameters is declared as follows:

```
def test(a, b: int, c: int = 1, d: int = 1) {
    return a + b + c + d
}
```

When the function *test* is invoked and the parameters `c` or `d` are not specified, the compiler will generate a call with default value to the unspecified parameter. A default parameter can be any valid compile-time const Daslang expression. The expression is evaluated at compile-time.

It is valid to declare default values for arguments other than the last one:

```
def test(c: int = 1, d: int = 1, a, b: int) { // valid!
    return a + b + c + d
}
```

Calling such functions with default arguments requires a named arguments call:

```
test(2, 3)           // invalid call, a,b parameters are missing
test([a = 2, b = 3]) // valid call
```

Default arguments can be combined with overloading:

```
def test(c: int = 1, d: int = 1, a, b: int) {
    return a + b + c + d
}
def test(a, b: int) { // now test(2, 3) is valid call
    return test([a = a, b = b])
}
```

2.8.2 OOP-style calls

There are no methods or function members of structs in Daslang. However, code can be easily written "OOP style" by using the right pipe operator |>:

```
struct Foo {
    x, y: int = 0
}

def setXY(var thisFoo: Foo; x, y: int) {
    thisFoo.x = x
    thisFoo.y = y
}
...
var foo:Foo
foo |> setXY(10, 11)   // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)     // exactly same as above line
```

(see *Structs*).

2.8.3 Tail Recursion

Tail recursion is a method for partially transforming recursion in a program into iteration: it applies when the recursive calls in a function are the last executed statements in that function (just before the return).

Currently, Daslang does not support tail recursion. A Daslang function is assumed to always return.

2.8.4 Operator Overloading

Daslang allows you to overload operators, which means that you can define custom behavior for operators when used with your own data types. To overload an operator, you need to define a special function with the name of the operator you want to overload. Here's the syntax:

```
def operator <operator>(<arguments>) : <return_type>
    # Implementation here
```

In this syntax, <operator> is the name of the operator you want to overload (e.g. +, -, *, /, ==, etc.), <arguments> are the parameters that the operator function takes, and <return_type> is the return type of the operator function.

For example, here's how you could overload the == operator for a custom struct called iVec2:

```
struct iVec2 {
    x, y: int
}

def operator==(a, b: iVec2) {
    return (a.x == b.x) && (a.y == b.y)
}
```

In this example, we define a structure called iVec2 with two integer fields (x and y).

We then define an operator== function that takes two parameters (a and b) of type iVec2. This function returns a bool value indicating whether a and b are equal. The implementation checks whether the x and y components of a and b are equal using the == operator.

With this operator overloaded, you can now use the == operator to compare iVec2 objects, like this:

```
let v1 = iVec2(1, 2)
let v2 = iVec2(1, 2)
let v3 = iVec2(3, 4)

print("{v1==v2}") // prints "true"
print("{v1==v3}") // prints "false"
```

In this example, we create three iVec2 objects and compare them using the == operator. The first comparison (v1 == v2) returns true because the x and y components of v1 and v2 are equal. The second comparison (v1 == v3) returns false because the x and y components of v1 and v3 are not equal.

2.8.5 Overloadable operators

The following table lists all operators that can be overloaded in Daslang:

Category	Operators
Arithmetic	+ - * / %
Comparison	== != < > <= >=
Bitwise	& | ^ ~ << >> <<< >>>
Logical	&& || ^^ !
Unary	- (negate) ~ (complement) ++ --
Compound assignment	+= -= *= /= %= &= |= ^= <<= >>= <<<= >>>= &&= ||= ^^=
Index	[] []= []<- []:= []+= []-= []*= etc.
Safe index	?[]
Dot	.?. . name . name := . name += etc.
Type	:= (clone) delete (finalize) is as ?as
Null coalesce	??
Interval	..

Operators can be defined as free functions or as struct methods.

2.8.6 Unary operators

Unary operators take a single argument. To overload unary minus (negate):

```
def operator -(a : Vec2) : Vec2 {
    return Vec2(x = -a.x, y = -a.y)
}
```

Prefix increment (++x) and postfix increment (x++) are separate operators. In the parser, ++operator is the prefix form and operator++ is the postfix form:

```
struct Counter {
    value : int
}

def operator ++(var c : Counter) : Counter {
    c.value += 1
    return c
}
```

The same pattern applies to --.

2.8.7 Compound assignment operators

Compound assignment operators modify the left-hand operand in place. The first parameter must be a mutable reference (var ... &):

```
def operator +=(var a : Vec2&; b : Vec2) {
    a.x += b.x
    a.y += b.y
}
```

(continues on next page)

(continued from previous page)

```
def operator *=(var a : Vec2&; s : float) {
    a.x *= s
    a.y *= s
}
```

This pattern works for all compound assignments: -=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=, &&=, ||=, ^^=.

2.8.8 Index operators

Index operators control how [] behaves on your types.

operator [] defines read access, operator []= defines write access, and compound variants like operator []+= define in-place index operations:

```
struct Matrix2x2 {
    data : float[4]
}

def operator [](m : Matrix2x2; i : int) : float {
    return m.data[i]
}

def operator []=(var m : Matrix2x2&; i : int; v : float) {
    m.data[i] = v
}

def operator []+=(var m : Matrix2x2&; i : int; v : float) {
    m.data[i] += v
}
```

Additional index operators include []<- (move into index), []:= (clone into index), []-=, []*=, and others matching the compound assignment family.

The safe index operator ?[] can be overloaded to return a default value when the index is out of range.

2.8.9 Clone and finalize operators

operator := overloads clone behaviour:

```
struct Resource {
    name : string
    refcount : int
}

def operator :=(var dst : Resource&; src : Resource) {
    dst.name = src.name
    dst.refcount = src.refcount + 1
}
```

Custom finalization can be defined via a finalize function or operator delete:

```
def finalize(var r : Resource) {
    print("releasing {r.name}\n")
}
```

2.8.10 is, as, and ?as operators

The `is`, `as`, and `?as` operators can be overloaded for custom type-checking and casting behaviour:

```
def operator is(a : MyVariant; b : type<int>) : bool {
    // return true if MyVariant currently holds an int
}

def operator as(a : MyVariant; b : type<int>) : int {
    // extract int value, panic if wrong type
}

def operator ?as(a : MyVariant; b : type<int>) : int? {
    // extract int value or return null
}
```

These are commonly used with variant types and in libraries like `daslib/ast_boost` and `daslib/json_boost`.

2.8.11 Null-coalesce operator

`operator ??` can be overloaded to provide a default value when a nullable or optional type is null:

```
def operator ??(a : MyOptional; default_value : int) : int {
    // return contained value or default_value
}
```

2.8.12 Struct method operators

Operators can be defined as struct methods instead of free functions:

```
struct Stack {
    items : array<int>

    def const operator [](index : int) : int {
        return items[index]
    }

    def operator []=(index : int; value : int) {
        items[index] = value
    }
}
```

Use the `const` qualifier on read-only operators. Write operators omit `const` because they mutate the struct's state.

2.8.13 Overloading the '.' and '?.' operators

Daslang allows you to overload the dot . operator, which is used to access fields of structure or a class. To overload the dot . operator, you need to define a special function with the name operator . Here's the syntax:

```
def operator.(<object>: <type>, <name>: string) : <return_type>
    # Implementation here
```

Alternatively you can specify field explicitly:

```
def operator.<name> (<object>: <type>) : <return_type>
    # Implementation here
```

In this syntax, <object> is the object you want to access, <type> is the type of the object, <name> is the name of the field you want to access, and <return_type> is the return type of the operator function.

Operator ?. works in a similar way.

For example, here's how you could overload the dot . operator for a custom structure called Goo:

```
struct Goo {
    a: string
}
def operator.(t: Goo, name: string) : string {
    return "{name} = {t . . a}"
}
def operator. length(t: Goo) : int {
    return length(t . . a)
}
```

In this example, we define a struct called Goo with a string field called a.

We then define two operator. functions:

The first one takes two parameters (t and name) and returns a string value that contains the name of the field or method being accessed (name) and the value of the a field of the Goo object (t.a). The second one takes one parameter (t) and returns the length of the a field of the Goo object (t.a). With these operators overloaded, you can now use the dot . operator to access fields and methods of a Goo object, like this:

```
var g = Goo(a ="hello")
var field = g.a
var length = g.length
```

In this example, we create an instance of the Goo struct and access its world field using the dot . operator. The overloaded operator. function is called and returns the string "world = hello". We also access the length property of the Goo object using the dot . operator. The overloaded operator. length function is called and returns the length of the a field of the Goo object (5 in this case).

The . . syntax is used to access the fields of a structure or a class while bypassing overloaded operations.

2.8.14 Overloading accessors

Daslang allows you to overload accessors, which means that you can define custom behavior for accessing fields of your own data types. Here is an example of how to overload the accessor for a custom struct called Foo:

```
require math

struct Foo {
    dir : float3
}
def operator . magnitude ( foo : Foo ) : float {
    return length(foo.dir)
}
def operator . magnitude := ( var foo:Foo; value:float ) {
    foo.dir = normalize(foo.dir) * value
}
[export]
def main {
    var f = Foo(dir=float3(1,2,3))
    print("magnitude = {f.magnitude} // {f}\n")
    f.magnitude := 10.
    print("magnitude = {f.magnitude} // {f}\n")
}
```

Expected output:

```
magnitude = 3.7416575 // [[ 1,2,3]]
magnitude = 10 // [[ 2.6726124,5.345225,8.017837]]
```

It now has accessor `magnitude` which can be used to get and set the magnitude of the `dir` field.

Classes allow to overload accessors for properties as well:

```
class Foo {
    dir : float3
    def const operator . magnitude : float {
        return length(dir)
    }
    def operator . magnitude := ( value:float ) {
        dir = normalize(dir) * value
    }
}
```

See also:

Generic programming for generic functions and type inference, *Lambdas* for anonymous callable objects with captures, *Blocks* for stack-bound callable objects, *Annotations* for function annotations like `[export]` and `[private]`, *Move, copy, and clone* for `return <-` move semantics, *Classes* for member functions and method-like calls.

2.9 Block

Blocks are nameless functions that capture the local context by reference. Blocks offer significant performance advantages over lambdas (see *Lambda*).

The block type can be declared with a function-like syntax. The type is written as `block` followed by an optional type signature in angle brackets:

```
block < (arg1:int; arg2:float&) : bool >
```

The `->` operator can be used instead of `:` for the return type:

```
block < (arg1:int; arg2:float&) -> bool >   // equivalent
```

If no type signature is specified, `block` alone represents a block that takes no arguments and returns nothing.

Blocks capture the current stack, so blocks can be passed, but never returned. Block variables can only be passed as arguments. Global or local block variables are prohibited; returning the block type is also prohibited:

```
def goo ( b : block )
    ...

def foo ( b : block < (arg1:int, arg2:float&) : bool >
    ...
```

Blocks can be called via `invoke`:

```
def radd(var ext:int&;b:block<(var arg:int&):int>):int {
    return invoke(b,ext)
}
```

There is also a shorthand, where block can be called as if it was a function:

```
def radd(var ext:int&;b:block<(var arg:int&):int>):int {
    return b(ext)   // same as invoke(b,ext)
}
```

Typeless blocks are typically declared via construction-like syntax:

```
goo() {                                 // block without arguments
    print("inside goo")
}
```

Similarly typed blocks are typically declared via pipe syntax:

```
var v1 = 1                               // block with arguments
res = radd(v1) $(var a:int&):int {
    return a++
}
```

Blocks can also be declared via inline syntax:

```
res = radd(v1, $(var a:int&) : int { return a++; }) // equivalent to example above
```

There is a simplified syntax for blocks that only contain a return expression:

```
res = radd(v1, $(var a:int&) : int => a++ )          // equivalent to example above
```

If a block is sufficiently specified in the generic or function, block types will be automatically inferred:

```
res = radd(v1, $(a) => a++ )                         // equivalent to example above
```

Nested blocks are allowed:

```
def passthroughFoo(a:Foo; blk:block<(b:Foo):void>) {
    invoke(blk,a)
}

passthrough(1) $(a) {
    assert(a==1)
    passthrough(2) $(b) {
        assert(a==1 && b==2)
        passthrough(3) $(c) {
            assert(a==1 && b==2 && c==3)
        }
    }
}
```

Loop control expressions are not allowed to cross block boundaries:

```
while ( true ) {
    take_any() {
        break               // 30801, captured block can't break outside the block
    }
}
```

Blocks can have annotations:

```
def queryOne(dt:float=1.0f) {
    testProfile::queryEs() $ [es] (var pos:float3&;vel:float3 const) { // [es] is␣
↪annotation
        pos += vel * dt
    }
}
```

Block annotations can be implemented via appropriate macros (see *Macro*).

Local block variables are allowed:

```
var blk = $ ( a, b : int ) {
    return a + b
}
verify ( 3 == invoke(blk,1,2) )
verify ( 7 == invoke(blk,3,4) )
```

Local block variables cannot be copied or moved.

See also:

Functions for regular function declarations, *Lambdas* for heap-allocated callable objects with captures, *Generators* for iterator-producing lambdas, *Annotations* for block annotations.

2.10 Lambda

Lambdas are nameless functions which capture the local context by clone, copy, or reference. Lambdas are slower than blocks, but allow for more flexibility in lifetime and capture modes (see *Blocks*).

The lambda type can be declared with a function-like syntax. The type is written as `lambda` followed by an optional type signature in angle brackets:

```
lambda < (arg1:int; arg2:float&) : bool >
```

The -> operator can be used instead of : for the return type:

```
lambda < (arg1:int; arg2:float&) -> bool >   // equivalent
```

If no type signature is specified, `lambda` alone represents a lambda that takes no arguments and returns nothing.

Lambdas can be local or global variables, and can be passed as an argument by reference. Lambdas can be moved, but can't be copied or cloned:

```
def foo ( x : lambda < (arg1:int;arg2:float&):bool > ) {
    ...
    var y <- x
    ...
}
```

Lambdas can be invoked via `invoke` or call-like syntax:

```
def inv13 ( x : lambda < (arg1:int):int > ) {
    return invoke(x,13)
}

def inv14 ( x : lambda < (arg1:int):int > ) {
    return x(14)
}
```

Lambdas are typically declared via move syntax:

```
var CNT = 0
let counter <- @ (extra:int) : int {
    return CNT++ + extra
}
let t = invoke(counter,13)
```

There are many similarities between lambda and block declarations. The main difference is that blocks are specified with $ symbol, while lambdas are specified with @ symbol. Lambdas can also be declared via inline syntax. There is a similar simplified syntax for the lambdas containing return expression only. If a lambda is sufficiently specified in the generic or function, its types will be automatically inferred (see *Blocks*).

2.10.1 Capture

Unlike blocks, lambdas can specify their capture types explicitly. There are several available types of capture:

- by copy (=)
- by move (<-)
- by clone (:=)
- by reference (&)

Capturing by reference requires unsafe.

By default, capture by copy is used. If copy is not available, the `unsafe` keyword is required for the default capture by move:

```
    var a1 <- [1,2]
    var a2 <- [1,2]
    var a3 <- [1,2]
    unsafe { // required do to capture of a1 by reference
            var lam <- @  capture(ref(a1),move(a2),clone(a3)) {
                    push(a1,1)
                    push(a2,1)
                    push(a3,1)
    }
            invoke(lam)
}
```

Lambdas can be deleted, which causes finalizers to be called on all captured data (see *Finalizers*):

```
delete lam
```

Lambdas can specify a custom finalizer which is invoked before the default finalizer:

```
var CNT = 0
var counter <- @ (extra:int) : int {
    return CNT++ + extra
} finally {
    print("CNT = {CNT}\n")
}
var x = invoke(counter,13)
delete counter                  // this is when the finalizer is called
```

2.10.2 Iterators

Lambdas are the main building blocks for implementing custom iterators (see *Iterators*).

Lambdas can be converted to iterators via the `each` or `each_ref` functions:

```
var count = 0
let lam <- @ (var a:int &) : bool {
    if ( count < 10 ) {
        a = count++
        return true
    } else {
```

(continues on next page)

(continued from previous page)

```
        return false
    }
}
for ( x,tx in each(lam),range(0,10) ) {
    assert(x==tx)
}
```

To serve as an iterator, a lambda must

- have single argument, which is the result of the iteration for each step
- have boolean return type, where `true` means continue iteration, and `false` means stop

A more straightforward way to make iterator is with generators (see *Generators*).

2.10.3 Implementation details

Lambdas are implemented by creating a nameless structure for the capture, as well as a function for the body of the lambda.

Let's review an example with a singled captured variable:

```
var CNT = 0
let counter <- @ (extra:int) : int {
    return CNT++ + extra
}
```

Daslang will generated the following code:

Capture structure:

```
struct _lambda_thismodule_7_8_1 {
    __lambda : function<(__this:_lambda_thismodule_7_8_1;extra:int const):int> = @@_
↪lambda_thismodule_7_8_1`function
    __finalize : function<(__this:_lambda_thismodule_7_8_1? -const):void> = @@_lambda_
↪thismodule_7_8_1`finalizer
    CNT : int
}
```

Body function:

```
def _lambda_thismodule_7_8_1`function ( var __this:_lambda_thismodule_7_8_1; extra:int␣
↪const ) : int {
    with ( __this ) {
        return CNT++ + extra
    }
}
```

Finalizer function:

```
def _lambda_thismodule_7_8_1`finalizer ( var __this:_lambda_thismodule_7_8_1? explicit )
↪{
    delete *this
    delete __this
}
```

Lambda creation is replaced with the ascend of the capture structure:

```
let counter:lambda<(extra:int const):int> const <- new<lambda<(extra:int const):int>>␣
↪(CNT = CNT)
```

The C++ Lambda class contains single void pointer for the capture data:

```
struct Lambda {
    ...
    char *      capture;
    ...
};
```

The rationale behind passing lambdas by reference is that when `delete` is called:

1. the finalizer is invoked for the capture data
2. the capture is replaced with null

The lack of copy or move semantics ensures that multiple pointers to a single instance of captured data cannot exist.

See also:

Blocks for stack-bound callable objects without captures, *Generators* for iterator-producing lambdas, *Functions* for regular named functions, *Unsafe* for implicit capture by reference or move, *Finalizers* for lambda finalizers.

2.11 Generator

Generators allow you to declare a lambda that behaves like an iterator. Internally, a generator is compiled into a lambda that is passed to an `each` or `each_ref` function.

Generator syntax is similar to lambda syntax. A generator expression starts with the `generator` keyword, followed by the element type in angle brackets, then a `$` and a block:

```
// generator<ElementType>() <| $ { ... }
```

Generator lambdas must have no arguments. The generator body always returns a boolean:

```
let gen <- generator<int>() <| $ {  // gen is iterator<int>
    for ( t in range(0,10) ) {
        yield t
    }
    return false                    // returning false stops iteration
}
```

The result type of a `generator` expression is an iterator (see *Iterators*).

Generators output iterator values via `yield` expressions. Similar to the return statement, move semantic `yield <-` is allowed:

```
return <- generator<TT> () <| $ {
    for ( w in src ) {
        yield <- invoke(blk,w)  // move invoke result
    }
    return false
}
```

Generators can output ref types. They can have a capture section:

```
unsafe {                                                    // unsafe due to capture of src␣
↪by reference
    var src = [1,2,3,4]
    var gen <- generator<int&> capture(ref(src)) () <| $ {      // capturing src by ref
        for ( w in src ) {
            yield w                                         // yield of int&
        }
        return false
    }
    for ( t in gen ) {
        t ++
    }
    print("src = {src}\n")  // will output [[2;3;4;5]]
}
```

Generators can have loops and other control structures:

```
let gen <- generator<int>() <| $ {
    var t = 0
    while ( t < 100 ) {
        if ( t == 10 ) {
            break
        }
        yield t ++
    }
    return false
}

let gen <- generator<int>() <| $ {
    for ( t in range(0,100) ) {
        if ( t >= 10 ) {
            continue
        }
        yield t
    }
    return false
}
```

Generators can have a `finally` expression on its blocks, with the exception of the if-then-else blocks:

```
let gen <- generator<int>() <| $ {
    for ( t in range(0,9) ) {
        yield t
    } finally {
        yield 9
    }
    return false
}
```

2.11.1 implementation details

In the following example:

```
var gen <- generator<int> () <| $ {
    for ( x in range(0,10) ) {
        if ( (x & 1)==0 ) {
            yield x
        }
    }
    return false
}
```

A lambda is generated with all captured variables:

```
struct _lambda_thismodule_8_8_1 {
    __lambda : function<(__this:_lambda_thismodule_8_8_1;_yield_8:int&):bool const> = @@_
↪::_lambda_thismodule_8_8_1`function
    __finalize : function<(__this:_lambda_thismodule_8_8_1? -const):void> = @@_::_lambda_
↪thismodule_8_8_1`finalizer
    __yield : int
    _loop_at_8 : bool
    x : int // captured constant
    _pvar_0_at_8 : void?
    _source_0_at_8 : iterator<int>
}
```

A lambda function is generated:

```
[GENERATOR]
def _lambda_thismodule_8_8_1`function ( var __this:_lambda_thismodule_8_8_1; var _yield_
↪8:int& ) : bool const {
    goto __this.__yield
    label 0:
    __this._loop_at_8 = true
    __this._source_0_at_8 <- __::builtin`each(range(0,10))
    memzero(__this.x)
    __this._pvar_0_at_8 = reinterpret<void?> addr(__this.x)
    __this._loop_at_8 &&= _builtin_iterator_first(__this._source_0_at_8,__this._pvar_0_
↪at_8,__context__)
    label 3: /*begin for at line 8*/
    if ( !__this._loop_at_8 ) {
            goto label 5
    }
    if ( !((__this.x & 1) == 0) ) {
            goto label 2
    }
    _yield_8 = __this.x
    __this.__yield = 1
    return /*yield*/ true
    label 1: /*yield at line 10*/
    label 2: /*end if at line 9*/
    label 4: /*continue for at line 8*/
    __this._loop_at_8 &&= _builtin_iterator_next(__this._source_0_at_8,__this._pvar_0_at_
```

(continues on next page)

(continued from previous page)

```
↪8,__context__)
    goto label 3
    label 5: /*end for at line 8*/
    _builtin_iterator_close(__this._source_0_at_8,__this._pvar_0_at_8,__context__)
    return false
}
```

Control flow statements are replaced with the `label` + `goto` equivalents. Generators always start with `goto __this.yield`. This effectively produces a finite state machine, with the `yield` variable holding current state index.

The `yield` expression is converted into a copy result and return value pair. A label is created to specify where to go to next time, after the `yield`:

```
_yield_8 = __this.x                 // produce next iterator value
__this.__yield = 1                  // label to go to next (1)
return /*yield*/ true               // return true to indicate, that iterator produced a␣
↪value
label 1: /*yield at line 10*/       // next label marker (1)
```

Iterator initialization is replaced with the creation of the lambda:

```
var gen:iterator<int> <- each(new<lambda<(_yield_8:int&):bool const>> default<_lambda_
↪thismodule_8_8_1>)
```

See also:

Lambdas for lambda capture semantics used by generators, *Statements* for `yield` and `return` statements, *Comprehensions* for iterator comprehensions backed by generators, *Blocks* for block-like syntax.

2.12 Struct

Daslang uses a structure mechanism similar to languages like C/C++, Java, C#, etc. However, there are some important differences. Structures are first-class objects like integers or strings and can be stored in table slots, other structures, local variables, arrays, tuples, variants, etc., and passed as function parameters.

2.12.1 Struct Declaration

A structure object is created through the keyword `struct`:

```
struct Foo {
    x, y: int
    xf: float
}
```

Structures can be `private` or `public`:

```
struct private Foo {
    x, y: int
}

struct public Bar {
```

(continues on next page)

(continued from previous page)

```
    xf: float
}
```

If not specified, structures inherit module publicity (i.e. in public modules structures are public, and in private modules structures are private).

Structure instances are created through a 'new expression' or a variable declaration statement:

```
let foo: Foo
let foo: Foo? = new Foo()
```

Structures intentionally have no member functions — only data members — since they represent a pure data type. Structures can contain members of function type as data (that is, function pointers that can be reassigned at runtime). Initializers simplify complex structure initialization. A function with the same name as the structure acts as its initializer. The compiler will generate a 'default' initializer if there are any members with an initializer:

```
struct Foo {
    x: int = 1
    y: int = 2
}
```

Structure fields are initialized to zero by default, regardless of 'initializers' for members, unless you specifically call the initializer:

```
let fZero : Foo     // no initializer is called, x, y = 0
let fInited = Foo() // initializer is called, x = 1, y = 2
```

Structure field types are inferred where possible:

```
struct Foo {
    x = 1    // inferred as int
    y = 2.0    // inferred as float
}
```

Explicit structure initialization during creation leaves all unspecified members zeroed:

```
let fExplicit = Foo(uninitialized x=13)  // x = 13, y = 0
```

The previous code example is syntactic sugar for:

```
let fExplicit: Foo
fExplicit.x = 13
```

Post-construction initialization only needs to specify overwritten fields:

```
let fPostConstruction = Foo(x=13)  // x = 13, y = 2
```

The previous code example is syntactic sugar for:

```
let fPostConstruction: Foo
fPostConstruction.x = 13
fPostConstruction.y = 2
```

The "Clone initializer" is a useful pattern for creating a clone of an existing structure when both structures are on the heap:

```
def Foo ( p : Foo? ) {                  // "clone initializer" takes pointer to existing␣
↪structure
    var self := *p
    return <- self
}
...
let a = new Foo(x=1, y=2.)              // create new instance of Foo on the heap,␣
↪initialize it
let b = new Foo(a)                      // clone of b is created here
```

2.12.2 Structure Function Members

Daslang doesn't have embedded structure member functions, virtual (that can be overridden in inherited structures) or non-virtual. Those features are implemented for classes. For ease of Objected Oriented Programming, non-virtual member functions can be easily emulated with the pipe operator |>:

```
struct Foo {
    x, y: int = 0
}

def setXY(var self: Foo; X, Y: int) {
    with ( self ) {
        x = X
        y = Y
    }
}

var foo: Foo
foo |> setXY(10, 11)   // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)     // exactly same thing as the line above
```

Since function pointers are first-class values, you can emulate virtual functions by storing function pointers as members:

```
struct Foo {
    x, y: int = 0
    set = @@setXY
}

def setXY(var self: Foo; X, Y: int) {
    with ( self ) {
        x = X
        y = Y
    }
}
...
var foo: Foo = Foo()
foo->set(1, 2)  // this one can call something else, if overridden in derived class.
                // It is also just syntactic sugar for function pointer call
invoke(foo.set, foo, 1, 2)  // exactly same thing as above
```

This makes the distinction between virtual and non-virtual calls in the OOP paradigm explicit and intentional. In fact, Daslang classes implement virtual functions in exactly this manner.

Virtual functions can be declared in the body of the structure. This is equivalent to the example above:

```
struct Foo {
    x, y: int = 0
    def setXY(X, Y: int) {
        x = X
        y = Y
    }
}
```

2.12.3 Inheritance

Daslang's structures support single inheritance by adding a ' : ', followed by the parent structure's name in the structure declaration. The syntax for a derived struct is the following:

```
struct Bar: Foo {
    yf: float
}
```

When a derived structure is declared, Daslang first copies all base's members to the new structure and then proceeds with evaluating the rest of the declaration.

A derived structure has all members of its base structure. It is just syntactic sugar for copying all the members manually first.

Virtual functions can be overridden in the derived structure:

```
struct Foo {
    x, y: int = 0
    def setXY(X, Y: int) {
        x = X
        y = Y
    }
}

struct Bar: Foo {
    yf: float = 0.0
    def override setXY(X, Y: int) {
        x = X + 1
        y = Y + 1
        yf = x + y
    }
}
```

2.12.4 Alignment

Structure size and alignment are similar to that of C++:

- individual members are aligned individually
- overall structure alignment is that of the largest member's alignment

Inherited structure alignment can be controlled via the `[cpp_layout]` annotation (see *Annotations*):

```
[cpp_layout (pod=false)]
struct CppS1 {
    vtable : void?              // we are simulating C++ class
    b : int64 = 2l
    c : int = 3
}

[cpp_layout (pod=false)]
struct CppS2 : CppS1 {          // d will be aligned on the class bounds
    d : int = 4
}
```

2.12.5 OOP implementation details

There is sufficient infrastructure to support basic OOP on top of structures. However, it is already available in form of classes with some fixed memory overhead (see *Classes*).

It's possible to override the method of the base class with override syntax. Here an example:

```
struct Foo {
    x, y: int = 0
    set = @@Foo_setXY
}

def Foo_setXY(var this: Foo; x, y: int) {
    this.x = x
    this.y = y
}

struct Foo3D: Foo {
    z: int = 3
    override set = cast<auto> @@Foo3D_setXY
}

def Foo3D_setXY(var thisFoo: Foo3D; x, y: int) {
    thisFoo.x = x
    thisFoo.y = y
    thisFoo.z = -1
}
```

It is safe to use the `cast` keyword to cast a derived structure instance into its parent type:

```
var f3d: Foo3D = Foo3D()
(cast<Foo> f3d).y = 5
```

It is unsafe to cast a base struct to its derived child type:

```
var f3d: Foo3D = Foo3D()
def foo(var foo: Foo) {
    (cast<Foo3D> foo).z = 5  // error, won't compile
}
```

If needed, the upcast can be used with the `unsafe` keyword:

```
struct Foo {
    x: int
}

struct Foo2:Foo {
    y: int
}

def setY(var foo: Foo; y: int) {  // Warning! Can cause problems if foo is not actually␣
↪Foo2
    unsafe {
        (upcast<Foo2> foo).y = y
    }
}
```

As the example above is very dangerous, and in order to make it safer, you can modify it to following:

```
struct Foo {
    x: int
    typeTag: uint = hash("Foo")
}

struct Foo2:Foo {
    y: int
    override typeTag: uint = hash("Foo2")
}

def setY(var foo: Foo; y: int) {  // this won't do anything really bad, but will panic on␣
↪wrong reference
    unsafe {
        if ( foo.typeTag == hash("Foo2") ) {
            (upcast<Foo2> foo).y = y
            print("Foo2 type references was passed\n")
        } else {
            assert(false, "Not Foo2 type references was passed\n")
        }
    }
}
```

See also:

Classes for the class type with member functions and RTTI, *Move, copy, and clone* for struct copy, move, and clone rules, *Finalizers* for struct finalization, *Functions* for OOP-style |> pipe calls on structs, *Generic programming* for `typeinfo` on structs.

2.13 Class

In Daslang, classes are an extension of structures designed to provide OOP capabilities. Classes provide single-parent inheritance, abstract and virtual methods, initializers, and finalizers.

The basic class declaration is similar to that of a structure, but with the `class` keyword:

```
class Foo {
    x, y : int = 0
    def Foo {                           // custom initializer
        Foo`set(self,1,1)
    }
    def set(X,Y:int) {                  // inline method
        x = X
        y = Y
    }
}
```

2.13.1 Member Visibility

Class members can be `private` or `public`. Private members are only accessible from inside the class:

```
class Account {
    private balance : int = 0
    def public deposit(amount : int) {
        balance += amount
    }
    def public get_balance : int {
        return balance
    }
}
```

Methods can also be declared `private`:

```
class Processor {
    def private validate(x : int) : bool {
        return x > 0
    }
    def process(x : int) {
        if (validate(x)) {
            print("{x}\n")
        }
    }
}
```

2.13.2 Initializers

The initializer is a function with a name matching that of the class. Classes can have multiple initializers with different arguments:

```
class Foo {
    ...
    def Foo(T:int) {                    // custom initializer
        set(T,T)
    }
    def Foo(X,Y:int) {                  // custom initializer
        Foo`set(self,X,Y)
    }
}
```

Finalizers can be defined explicitly as void functions named `finalize`:

```
class Foo {
    ...
    def finalize {                      // custom finalizer
        delFoo ++
    }
}
```

An alternative syntax for finalizers is:

```
class Foo {
    ...
    def operator delete {               // custom finalizer
        delFoo ++
    }
}
```

There are no guarantees that a finalizer is called implicitly (see *Finalizers*).

Derived classes need to override methods explicitly, using the `override` keyword:

```
class Foo3D : Foo {
    z : int = 13
    def Foo3D {                         // overriding default initializer
        Foo`Foo(self)                   // call parents initializer explicitly
        z = 3
    }
    def override set(X,Y:int) {         // overriding parent method
        Foo`set(self,X,Y)               // calling generated method function directly
        z = 0
    }
}
```

2.13.3 Calling Parent Methods

Inside a derived class, `super()` calls the parent class constructor:

```
class Derived : Base {
    def Derived {
        super()             // calls Base's default initializer
        z = 3
    }
}
```

`super.method(args)` calls a specific parent method, bypassing the virtual dispatch:

```
class Derived : Base {
    def override process(x : int) {
        super.process(x)    // calls Base`process, not the overridden version
        // additional logic
    }
}
```

Both forms are rewritten by the compiler into explicit calls to the parent class function: `super()` becomes `Base`Base(self)` and `super.process(x)` becomes `Base`process(self, x)`.

The option `always_call_super` can be enabled to require `super()` in every constructor (see *Options*).

Alternatively, the parent's method can be called directly using the backtick syntax:

```
Foo`set(self, X, Y)         // equivalent to super.set(X, Y) from within Foo3D
```

Classes can define abstract methods using the `abstract` keyword:

```
class FooAbstract {
    def abstract set(X,Y:int) : void            // inline method
}
```

Abstract method declarations must be fully qualified, including their return type. Class member functions are inferred in the same manner as regular functions.

Sealed functions cannot be overridden. The `sealed` keyword is used to prevent overriding:

```
class Foo3D : Foo {
    def sealed set(X,Y:int ) { // subclasses of Foo3D can no longer override this method
        xyz = X + Y
    }
}
```

Sealed classes can not be inherited from. The `sealed` keyword is used to prevent inheritance:

```
class sealed Foo3D : Foo        // Foo3D can no longer be inherited from
    ...
```

A pointer named `self` is available inside any class method. Because the method body is wrapped in `with(self)`, all fields and methods can be accessed directly — `self.` is not required:

```
class Foo {
    x : int
```

(continues on next page)

(continued from previous page)

```
    def set(val : int) {
        x = val             // same as self.x = val
    }
}
```

Classes can be created via the new operator:

```
var f = new Foo()
```

Local class variables are unsafe:

```
unsafe {
    var f = Foo()       // unsafe
}
```

Class methods can be invoked using . syntax:

```
f.set(1,2)
```

The -> operator can also be used:

```
f->set(1,2)
```

A specific version of the method can also be called explicitly:

```
Foo`set(*f,1,2)
```

Class methods can be constant:

```
class Foo {
    dir : float3
    def const length {
        return length(dir)  // dir is const float3 here
    }
}
```

Class methods can be operators:

```
class Foo {
    dir : float3
    def Foo ( x,y,z:float ) {
        dir = float3(x,y,z)
    }
    def Foo ( d:float3 ) {
        dir = d
    }
    def const operator . length {
        return length(dir)
    }
    def operator . length := (value : float) {
        dir = normalize(dir) * value
    }
    def const operator + (other : Foo) : Foo {
```

(continues on next page)

(continued from previous page)

```
        return unsafe(Foo(dir + other.dir))
    }
}
```

Class fields can be declared static, i.e. shared between all instances of the class:

```
class Foo {
    static count : int = 0
    def Foo {
        count ++
    }
    def finalize {
        count --
    }
}
```

Class methods can be declared static. Static methods don't have access to 'self' but can access static fields:

```
class Foo {
    static count : int = 0
    def static getCount : int {
        return count
    }
}

    let count = Foo`getCount()  // they can be accessed outside of class
```

2.13.4 Runtime Type Checking (is)

The `is` operator checks whether a class instance is of a given type or any type derived from it. This requires the `daslib/dynamic_cast_rtti` module:

```
require daslib/dynamic_cast_rtti

class Animal {
    def abstract speak : string
}

class Dog : Animal {
    def override speak : string {
        return "woof"
    }
}

class Cat : Animal {
    def override speak : string {
        return "meow"
    }
}

var a : Animal? = new Dog()
```

(continues on next page)

(continued from previous page)

```
assert(a is Dog)            // true - a points to a Dog
assert(a is Animal)         // true - Dog is an Animal
assert(!(a is Cat))         // true - a is not a Cat
```

Without `daslib/dynamic_cast_rtti`, the `is` operator performs a static (compile-time) type check only. With the module, it performs runtime RTTI checking by walking the class hierarchy via the `__rtti` field.

2.13.5 Type Casting (as, ?as)

The `as` and `?as` operators cast a class pointer to a more specific type. These also require `daslib/dynamic_cast_rtti`.

`as` performs a forced cast — it panics if the cast fails:

```
var a : Animal? = new Dog()
var d = a as Dog            // succeeds: a is a Dog
d.speak()                   // "woof"
```

`?as` performs a safe cast — it returns `null` if the cast fails:

```
var a : Animal? = new Dog()
var c = a ?as Cat           // returns null: a is not a Cat
if (c != null) {
    c.speak()
}
```

Use `?as` when you are not certain of the runtime type. Use `as` when failure indicates a programming error.

Custom `operator is`, `operator as`, and `operator ?as` can also be defined for non-class types (see *Pattern Matching*).

2.13.6 Class Templates

A `class template` declaration creates a parameterized class pattern. Template classes are not instantiated directly — they serve as blueprints for code generation via macros:

```
[template_structure(KeyType, ValueType)]
class template TCache {
    keys : array<KeyType>
    values : array<ValueType>
    def lookup(key : KeyType) : ValueType? {
        for (k, v in keys, values) {
            if (k == key) {
                return unsafe(addr(v))
            }
        }
        return null
    }
}
```

Template parameters (`KeyType`, `ValueType`) are replaced with concrete types during instantiation. The `template_structure` annotation (from `daslib/typemacro_boost`) handles the substitution.

Template methods automatically inherit the template flag.

2.13.7 Static methods with [class_method]

The [class_method] annotation (from daslib/class_boost) turns a def static function inside a struct or class into a method with an implicit self argument. The macro automatically adds self as the first parameter and wraps the function body in with(self) { ... }, allowing direct access to fields without self. prefix.

This is the primary way to define methods on structs, since structs don't have built-in method dispatch like classes. It also works on classes when virtual dispatch is not needed.

Methods defined with [class_method] are called using dot syntax: obj.method().

def static — mutable method

The self parameter is mutable, so the method can modify fields:

```
require daslib/class_boost

struct Counter {
    value : int = 0

    [class_method]
    def static increment(amount : int = 1) {
        value += amount
    }

    [class_method]
    def static reset {
        value = 0
    }
}
```

def static const — const method

The self parameter is const, so the method cannot modify fields:

```
struct Counter {
    ...
    [class_method]
    def static const get_value : int {
        return value
    }
}
```

[explicit_const_class_method]

When both a const and a non-const overload of the same method are needed, use [explicit_const_class_method] instead of [class_method]. This is common for methods like foreach or operator [] that should work on both mutable and immutable instances:

```
require daslib/class_boost

struct Container {
    data : array<int>

    [explicit_const_class_method]
    def static foreach(blk : block<(x : int) : void>) {
        for (x in data) {
            blk |> invoke(x)
        }
    }

    [explicit_const_class_method]
    def static const foreach(blk : block<(x : int) : void>) {
        for (x in data) {
            blk |> invoke(x)
        }
    }
}
```

Complete [class_method] example

```
require daslib/class_boost

struct Counter {
    value : int = 0

    [class_method]
    def static increment(amount : int = 1) {
        value += amount
    }

    [class_method]
    def static const get_value : int {
        return value
    }

    [class_method]
    def static reset {
        value = 0
    }
}

[export]
def main {
    var c = Counter()
```

(continues on next page)

(continued from previous page)

```
    c.increment()
    c.increment(10)
    print("value = {c.get_value()}\n")
    c.reset()
    print("after reset = {c.get_value()}\n")
}
```

Expected output:

```
value = 11
after reset = 0
```

2.13.8 Stack-Allocated Classes (using)

The `using` construct creates a stack-allocated handle instance that is automatically finalized at the end of the scope:

```
using() $(var s : das_string) {
    s := "hello"
    print("{s}\n")
    // s is finalized here
}
```

This avoids a heap allocation and manual cleanup. The handle is constructed on the stack, and its finalizer runs when the block exits. The `using` function is provided by C++ handle types (such as `das_string`) through their factory bindings.

2.13.9 Complete Example

Here is a complete example demonstrating inheritance, abstract methods, custom initializers, and virtual dispatch:

```
class Shape {
    name : string
    def abstract area : float
    def describe {
        print("{name}: area = {area()}\n")
    }
}

class Circle : Shape {
    radius : float
    def Circle(r : float) {
        name = "Circle"
        radius = r
    }
    def override area : float {
        return 3.14159 * radius * radius
    }
}

class Rectangle : Shape {
```

(continues on next page)

(continued from previous page)

```
    width, height : float
    def Rectangle(w, h : float) {
        name = "Rectangle"
        width = w
        height = h
    }
    def override area : float {
        return width * height
    }
}

[export]
def main {
    var shapes : array<Shape?>
    unsafe {
        shapes |> push(new Circle(5.0))
        shapes |> push(new Rectangle(3.0, 4.0))
    }
    for (s in shapes) {
        s.describe()
    }
    unsafe {
        for (s in shapes) {
            delete s
        }
    }
}
```

Expected output:

```
Circle: area = 78.53975
Rectangle: area = 12
```

2.13.10 Implementation details

Class initializers are generated by adding a local `self` variable with `construct` syntax. The body of the method is prefixed via a `with self` expression. The final expression is a `return <- self`:

```
def Foo ( X:int const; Y:int const ) : Foo {
    var self:Foo <- Foo(uninitialized)
    with ( self ) {
        Foo`Foo(self,X,Y)
    }
    return <- self
}
```

Class methods and finalizers are generated by providing the extra argument `self`. The body of the method is prefixed with a `with self` expression:

```
def Foo3D`set ( var self:Foo3D; X:int const; Y:int const ) {
    with ( self ) {
```

(continues on next page)

(continued from previous page)

```
        Foo`set(self,X,Y)
        z = 0
    }
}
```

Calling virtual methods is implemented via invoke:

```
invoke(f3d.set,cast<Foo> f3d,1,2)
```

Every base class gets an `__rtti` pointer, and a `__finalize` function pointer. Additionally, a function pointer is added for each member function:

```
class Foo {
    __rtti : void? = typeinfo(rtti_classinfo type<Foo>)
    __finalize : function<(self:Foo):void> = @@_::Foo'__finalize
    x : int = 0
    y : int = 0
    set : function<(self:Foo;X:int const;Y:int const):void> = @@_::Foo`set
}
```

`__rtti` contains rtti::TypeInfo for the specific class instance. There is helper function in the rtti module to access class_info safely:

```
def class_info ( cl ) : StructInfo const?
```

The `finalize` pointer is invoked when the finalizer is called for the class pointer. That way, when delete is called on the base class pointer, the correct version of the derived finalizer is called.

See also:

Structs for the pure-data struct type without member functions, *Move, copy, and clone* for class copy and move rules (unsafe), *Annotations* for class annotations, *Pattern matching* for `is`/`as`/`?as` with class hierarchies.

2.14 Tuple

Tuples are a concise syntax to create anonymous data structures. A tuple type is declared with the `tuple` keyword followed by a list of element types (optionally named) in angle brackets:

```
tuple<int; float>               // unnamed elements
tuple<i:int; f:float>           // named elements
```

Two tuple declarations are the same if they have the same number of types, and their respective types are the same:

```
var a : tuple<int, float>
var b : tuple<i:int, f:float>
a = b
```

Tuple elements can be accessed via nameless fields, i.e. _ followed by the 0 base field index:

```
a._0 = 1
a._1 = 2.0
```

Named tuple elements can be accessed by name as well as via nameless field:

```
b.i = 1         // same as _0
b.f = 2.0       // same as _1
b._1 = 2.0      // _1 is also available
```

Tuples follow the same alignment rules as structures (see *Structures*).

Tuple alias types can be constructed the same way as structures. For example:

```
tuple Foo {
    a : int
    b : float
}
```

It's the same as:

```
typedef Foo = tuple<a:int,b:float>
```

Tuples can be constructed using the tuple constructor, for example:

```
var a = (1,2.0,"3")
var b = tuple(1, 2.0, "3")
```

The => operator creates a 2-element tuple from its left and right operands:

```
var c = "one" => 1   // same as tuple<string,int>("one", 1)
```

This works in any expression context, not just table literals. Table literals like `{ "one"=>1, "two"=>2 }` use => to form key-value tuples that are then inserted into the table (see *Tables*).

Tuple elements can be assigned names via tuple constructor:

```
var a = tuple<a:int,b:float,c:string>(a=1, b=2.0, c="3")
```

Both `auto` and a full type specification can be used to construct a tuple. Array of tuples can be constructed using similar syntax, with a comma as a separator:

```
let H : array<Tup> <- array tuple<Tup>((a = 1, b = 2., c = "3"), (a = 4, b = 5., c = "6
↪"))
```

Tuples can be expanded upon the variable declaration, for example:

```
var (a, b, c) = (1, 2.0, "3")
```

In this case only one variable is created, as well as for 'assume' expressions. I.e:

```
var a`b`c = (1, 2.0, "3")
assume a  = a`b`c._0
assume b  = a`b`c._1
assume c  = a`b`c._2
```

Iterators and containers can be expanded in the for-loop in a similar way:

```
var H <- [(1, 2.0, "3"), (4, 5.0, "6")]
for ( (a, b, c) in H ) {
    assert(a == 1)
```

(continues on next page)

(continued from previous page)

```
    assert(b == 2.0)
    assert(c == "3")
}
```

See also:

Datatypes for a list of built-in types, *Pattern matching* for matching and destructuring tuples, *Finalizers* for tuple finalization, *Move, copy, and clone* for tuple copy and move rules, *Aliases* for the `typedef` shorthand tuple syntax.

2.15 Variant

Variants are nameless types which provide support for values that can be one of a number of named cases, possibly each with different values and types:

```
var t : variant<i_value:uint,f_value:float>
```

There is a shorthand type alias syntax to define a variant:

```
variant U_F {
    i_value : uint
    f_value : float
}

typedef U_F = variant<i_value:uint,f_value:float> // exactly the same as the declaration␣
↪above
```

Any two variants are the same type if they have the same named cases of the same types in the same order.

Variants hold the `index` of the current case, as well as the value for the current case only.

The current case selection can be checked via the `is` operator, and accessed via the `as` operator:

```
assert(t is i_value)
assert(t as i_value == 0x3f800000)
```

The entire variant selection can be modified by copying the properly constructed variant of a different case:

```
t = U_F(i_value = 0x40000000)    // now case is i_value
t = U_F(f_value = 1.0)           // now case is f_value
```

Accessing a variant case of the incorrect type will cause a panic:

```
t = U_F(i_value = 0x40000000)
return t as f_value                 // panic, invalid variant index
```

Safe navigation is available via the `?as` operation:

```
return t ?as f_value ?? 1.0         // will return 1.0 if t is not f_value
```

Cases can also be accessed in an unsafe manner without checking the type:

```
unsafe {
    t.i_value = 0x3f800000
```

(continues on next page)

(continued from previous page)

```
    return t.f_value                    // will return memory, occupied by f_value - i.e.
↪ 1.0f
}
```

The current index can be determined via the `variant_index` function:

```
var t : U_F
assert(variant_index(t)==0)
```

The index value for a specific case can be determined via the `variant_index` and `safe_variant_index` type traits. `safe_variant_index` will return -1 for invalid indices and types, whereas `variant_index` will report a compilation error:

```
assert(typeinfo(variant_index<i_value> t)==0)
assert(typeinfo(variant_index<f_value> t)==1)
assert(typeinfo(variant_index<unknown_value> t)==-1) // compilation error

assert(typeinfo(safe_variant_index<i_value> t)==0)
assert(typeinfo(safe_variant_index<f_value> t)==1)
assert(typeinfo(safe_variant_index<unknown_value> t)==-1)
```

Current case selection can be modified with the unsafe operation `safe_variant_index`:

```
unsafe {
    set_variant_index(t, typeinfo variant_index<f_value>(t))
}
```

2.15.1 Alignment and data layout

Variants contain the 'index' of the current case, followed by a union of individual cases, similar to the following C++ layout:

```
struct MyVariantName {
    int32_t __variant_index;
    union {
        type0   case0;
        type1   case1;
        ...
    };
};
```

Individual cases start from the same offset.

The variant type is aligned by the alignment of its largest case, but no less than that of an int32.

See also:

Pattern matching for matching and extracting variant cases, *Aliases* for the `typedef` shorthand variant syntax, *Datatypes* for a list of built-in types, *Move, copy, and clone* for variant clone and move semantics, *Unsafe* for unsafe variant field access and `?as` without null coalescing, *Finalizers* for variant finalization.

2.16 Bitfield

Bitfields are nameless types that represent a collection of flags in a single integer:

```
var t : bitfield < one,two,three >
```

There is a shorthand type alias syntax to define a bitfield:

```
bitfield bits123 {
    one
    two
    three
}

typedef bits123 = bitfield<one; two; three> // exactly the same as the declaration above
```

Bitfield flags can be 8, 16, 32 or 64 bits in size. By default, bitfields are 32 bits. To specify a different size, use `: bitfield : uintXX` syntax:

```
var t8  : bitfield : uint8 <one; two; three>
var t16 : bitfield : uint16 <one; two; three>
var t32 : bitfield : uint <one; two; three>
var t32_by_default : bitfield <one; two; three>
var t64 : bitfield : uint64 <one; two; three>
```

A type alias notation can also be used to specify the size:

```
bitfield bits123_8bit : uint8 {
    one
    two
    three
}

bitfield bits123_16bit : uint16 {
    one
    two
    three
}

bitfield bits123_32bit : uint {
    one
    two
    three
}

bitfield bits123_64bit : uint64 {
    one
    two
    three
}
```

Any two bitfields are the same type, if an underlying integer type is the same:

```
var a : bitfield<one; two; three>
var b : bitfield<one; two>
b = a
```

Individual flags can be read as if they were regular bool fields:

```
var t : bitfield < one; two; three >
assert(!t.one)
```

If an alias is available, a bitfield can be constructed using alias notation:

```
assert(t==bits123.three)
```

Bitfields can be constructed via an integer value. Limited binary logical operators are available:

```
var t : bitfield < one; two; three > = bitfield(1<<1) | bitfield(1<<2)
assert(!t.one && t.two && t.three)
assert("{t}"=="(two|three)")
t ^= bitfield(1<<1)
```

Bitfields support built-in constants:

```
bitfield OneTwoThree {
    one
    two
    three
    All = OneTwoThree.one | OneTwoThree.two | OneTwoThree.three
    None = bitfield(0)
}
```

See also:

Datatypes for a complete list of built-in types, *Aliases* for the `typedef` shorthand syntax, *Expressions* for binary operators used with bitfields.

2.17 Type Aliases

Type aliases allow you to give a new name to an existing type. This improves code readability and makes it easier to work with complex type declarations. Aliases are fully interchangeable with the types they refer to — they do not create new types.

2.17.1 typedef Declaration

A type alias is declared with the `typedef` keyword:

```
typedef Vec3 = float3
typedef StringArray = array<string>
typedef Callback = function<(x:int; y:int):bool>
```

Once defined, the alias can be used anywhere a type is expected:

```
var position : Vec3
var names : StringArray
var cb : Callback
```

Aliases can refer to complex types, including tables, arrays, tuples, variants, blocks, and lambdas:

```
typedef IntPair = tuple<int; int>
typedef Registry = table<string; array<int>>
typedef Transform = block<(pos:float3; vel:float3):float3>
```

2.17.2 Publicity

Type aliases can be `public` or `private`:

```
typedef public  Vec3 = float3       // visible to other modules
typedef private Internal = int      // only visible within this module
```

If no publicity is specified, the alias inherits the module's default publicity (i.e., in public modules aliases are public, and in private modules they are private).

2.17.3 Shorthand Type Alias Syntax

Several composite types support a shorthand declaration syntax that creates a named type alias.

Tuple Aliases

Instead of writing a `typedef` for a tuple, you can use the `tuple` keyword directly:

```
tuple Vertex {
    position : float3
    normal   : float3
    uv       : float2
}
```

This is equivalent to:

```
typedef Vertex = tuple<position:float3; normal:float3; uv:float2>
```

(see *Tuples*).

Variant Aliases

Variants support a similar shorthand:

```
variant Value {
    i : int
    f : float
    s : string
}
```

This is equivalent to:

```
typedef Value = variant<i:int; f:float; s:string>
```

(see *Variants*).

Bitfield Aliases

Bitfields also support the shorthand syntax:

```
bitfield Permissions {
    read
    write
    execute
}
```

This is equivalent to:

```
typedef Permissions = bitfield<read; write; execute>
```

Bitfield aliases can specify a storage type:

```
bitfield SmallFlags : uint8 {
    active
    visible
}
```

(see *Bitfields*).

2.17.4 Local Type Aliases

Type aliases can be declared inside function bodies:

```
def process {
    typedef Entry = tuple<string; int>
    var data : Entry
    data = ("hello", 42)
}
```

Local type aliases are scoped to the enclosing block and are not visible outside it.

Type aliases can also be declared inside structure or class bodies:

```
struct Container {
    typedef Element = int
    data : array<Element>
}
```

2.17.5 Generic Type Aliases (auto)

In generic functions, the `auto(Name)` syntax creates inferred type aliases. These are resolved during function instantiation based on the actual argument types:

```
def first_element(a : array<auto(ElementType)>) : ElementType {
    return a[0]
}

let x = first_element([1, 2, 3])    // ElementType is inferred as int, returns int
```

Named auto aliases can be reused across multiple arguments to enforce type relationships:

```
def add_to_array(var arr : array<auto(T)>; value : T) {
    arr |> push(value)
}
```

(see *Generic Programming*).

2.17.6 Type Aliases with typedecl

The `typedecl` expression can be used inside generic functions to create types based on the type of an expression:

```
def make_table(key : auto(K)) {
    var result : table<typedecl(key); string>
    return <- result
}
```

Here the key type of the table is inferred from the type of the `key` argument.

(see *Generic Programming*).

See also:

Datatypes for the built-in types that can be aliased, *Tuples* for tuple alias syntax, *Variants* for variant alias syntax, *Bitfields* for bitfield alias syntax.

2.18 Array

An array is a sequence of values indexed by an integer number from 0 to the size of the array minus 1. An array's elements can be obtained by their index:

```
var a = fixed_array<int>(1, 2, 3, 4) // fixed size of array is 4, and content is [1, 2,␣
↪3, 4]
assert(a[0] == 1)

var b: array<int>
push(b,1)
assert(b[0] == 1)
```

Alternative syntax is:

```
var a = fixed_array(1,2,3,4)
```

There are static arrays (of fixed size, allocated on the stack), and dynamic arrays (size is dynamic, allocated on the heap):

```
var a = fixed_array(1, 2, 3, 4) // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>            // empty dynamic array
push(b, "some")                 // now it is 1 element of "some"
b |> push("some")               // same as above line, but using pipe operator
```

Dynamic sub-arrays can be created out of any array type via range indexing:

```
var a  = fixed_array(1,2,3,4)
let b <- a[1..3]               //  b is [2,3]
```

In reality `a[b]`, where b is a range, is equivalent to `subarray(a, b)`.

Resizing, insertion, and deletion of dynamic arrays and array elements is done through a set of standard functions (see built-in functions).

The relevant builtin functions are: `push`, `push_clone`, `emplace`, `reserve`, `resize`, `erase`, `length`, `clear`, `empty` and `capacity`.

Arrays (as well as tables, structures, and handled types) are passed to functions by reference only.

Arrays cannot be copied; only cloned or moved.

```
def clone_array(var a, b: array<string>)
  a := b      // a is now a deep copy of b
  clone(a, b) // same as above

def move_array(var a, b: array<string>)
  a <- b  // a now points to the same data as b, and b is empty
```

Arrays can be constructed inline:

```
let arr = fixed_array(1.,2.,3.,4.5)
```

This expands to:

```
let arr : float[4] = fixed_array<float>(1.,2.,3.,4.5)
```

Dynamic arrays can also be constructed inline:

```
let arr <- ["one"; "two"; "three"]
```

This is syntactic equivalent to:

```
let arr : array<string> <- array<string>("one","two","three")
```

Alternative syntax is:

```
let arr <- array(1., 2., 3., 4.5)
let arr <- array<float>(1., 2., 3., 4.5)
```

Arrays of structures can be constructed inline:

```
struct Foo {
  x : int = 1
```

(continues on next page)

(continued from previous page)

```
  y : int = 2
}

var a <- array struct<Foo>((x=11,y=22),(x=33),(y=44))    // dynamic array of Foo with
↪'construct' syntax (creates 3 different copies of Foo)
```

Arrays of variants can be constructed inline:

```
variant Bar {
  i : int
  f : float
  s : string
}

var a <- array variant<Bar>(Bar(i=1), Bar(f=2.), Bar(s="3"))    // dynamic array of Bar␣
↪(creates 3 different copies of Bar)
```

Arrays of tuples can be constructed inline:

```
var a <- array tuple<i:int,s:string>((i=1,s="one"),(i=2,s="two"),(i=3,s="three"))    //␣
↪dynamic array of tuples (creates 3 different copies of tuple)
```

When array elements can't be copied, use `push_clone` to insert a clone of a value, or `emplace` to move it in.

`resize` can potentially create new array elements. Those elements are initialized with 0.

`reserve` is there for performance reasons. Generally, array capacity doubles, if exceeded. `reserve` allows you to specify the exact known capacity and significantly reduce the overhead of multiple `push` operations.

`empty` tells you if the dynamic array is empty.

`clear` clears the array, but does not free the memory. It is equivalent to `resize(0)`.

`length` returns the number of elements in the array.

`capacity` returns the number of elements that can be stored in the array without reallocating.

It's possible to iterate over an array via a regular `for` loop.

```
for ( x in [1,2,3,4] ) {
  print("x = {x}\n")
}
```

Additionally, a collection of unsafe iterators is provided:

```
def each ( a : auto(TT)[] ) : iterator<TT&>
def each ( a : array<auto(TT)> ) : iterator<TT&>
```

The reason both are unsafe operations is that they do not capture the array.

Search functions are available for both static and dynamic arrays:

```
def find_index ( arr : array<auto(TT)> implicit; key : TT )
def find_index ( arr : auto(TT)[] implicit; key : TT )
def find_index_if ( arr : array<auto(TT)> implicit; blk : block<(key:TT):bool> )
def find_index_if ( arr : auto(TT)[] implicit; blk : block<(key:TT):bool> )
```

See also:

Comprehensions for array and iterator comprehension syntax, *Iterators* for iteration patterns over arrays, *Move, copy, and clone* for array copy, move, and clone rules, *Locks* for array locking during iteration, *Datatypes* for a list of element types.

2.19 Table

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string,int>
unsafe {
    tab["10"] = 10
    tab["20"] = 20
    tab["some"] = 10
    tab["some"] = 20  // replaces the value for 'some' key
}
```

Accessing a table element via the index operator is unsafe, because Daslang containers store unboxed values. Consider the following example:

```
tab["1"] = tab["2"]              // this potentially breaks the table
```

What happens is table may get resized either after tab[“1”] or tab[“2”] if either key is missing (similar to C++ STL hash_map).

It is possible to suppress this unsafe error via `CodeOfPolicies`, or by using the following option:

```
options unsafe_table_lookup = false
```

Safe navigation of the table is safe, since it does not create missing keys:

```
var tab <- { "one"=>1, "two"=>2 }
let t = tab?["one"] ?? -1            // this is safe, since it does not create missing␣
↪keys
assert(t==1)
```

It can also be written to:

```
var tab <- { "one"=>1, "two"=>2 }
var dummy = 0
tab?["one"] ?? dummy = 10   // if "one" is not in the table, dummy will be set to 10
```

A collection of safe functions is available for working with tables, even if the table is empty or gets resized.

There are several relevant builtin functions: `clear`, `key_exists`, `get`, and `erase`. `get` works with block as last argument, and returns if value has been found. It can be used with the rbpipe operator:

```
let tab <- { "one"=>1, "two"=>2 }
let found = get(tab,"one") $(val) {
    assert(val==1)
}
assert(found)
```

A non-constant version is available as well:

```
var tab <- { "one"=>1, "two"=>2 }
let found = get(tab,"one") $(var val) {
    val = 123
}
let t = tab |> get_value("one")
assert(t==123)
```

insert, insert_clone, and emplace are safe ways to add elements to a table:

```
var tab <- { "one"=>1, "two"=>2 }
tab |> insert("three",3)          // insert new key/value pair
tab |> insert_clone("four",4)     // insert new key/value pair, but clone the value
var five = 5
tab |> emplace("five",five)       // insert new key/value pair, but move the value
```

Tables (as well as arrays, structs, and handled types) are passed to functions by reference only.

Tables cannot be assigned, only cloned or moved.

```
def clone_table(var a, b: table<string, int>) {
    a := b      // a is now a deep copy of b
    clone(a, b) // same as above
    a = b       // error
}

def move_table(var a, b: table<string, int>) {
    a <- b  // a now points to the same data as b, and b is empty
}
```

Table keys can be not only strings, but any other 'workhorse' type as well.

Tables can be constructed inline:

```
let tab <- { "one"=>1, "two"=>2 }
```

This is syntax sugar for:

```
let tab : table<string,int> <- to_table_move(fixed_array<tuple<string,int>>(("one",1),(
↪"two",2)))
```

Alternative syntax is:

```
let tab <- table("one"=>1, "two"=>2)
let tab <- table<string,int>("one"=>1, "two"=>2)
```

A table that holds no associative data can also be declared:

```
var tab : table<int>
tab |> insert(1)        // this is how we insert a key into such table
```

A table can be iterated over with the for loop:

```
var tab <- { "one"=>1, "two"=>2 }
for ( key, value in keys(tab), values(tab) ) {
    print("key: {key}, value: {value}\n")
}
```

A table that holds no associative data only has keys.

See also:

Datatypes for a list of key and value types, *Iterators* for `keys` and `values` iterator patterns, *Move, copy, and clone* for table move and clone semantics, *Locks* for table locking during iteration and find, *Unsafe* for `unsafe_table_lookup` and pointer-related operations, *Tuples* for the inline table construction syntax.

2.20 Iterator

Iterators are objects that traverse a sequence without exposing the details of the sequence's implementation.

The iterator type is written as `iterator` followed by the element type in angle brackets:

```
iterator<int>           // iterates over integers
iterator<const Foo&>    // iterates over Foo by const reference
```

Iterators can be moved, but not copied or cloned.

Iterators can be created via the `each` function from a range, static array, or dynamic array. `each` functions are unsafe because the iterator does not capture its arguments:

```
unsafe {
    var it <- each ( [1,2,3,4] )
}
```

The most straightforward way to traverse an iterator is with a `for` loop:

```
for ( x in it ) {             // iterates over contents of 'it'
    print("x = {x}\n")
}
```

For the reference iterator, the `for` loop will provide a reference variable:

```
var t = fixed_array(1,2,3,4)
for ( x in t ) {        // x is int&
    x ++                // increases values inside t
}
```

Iterators can be created from lambdas (see *Lambda*) or generators (see *Generator*).

2.20.1 Making types iterable

Any type can be made directly iterable in a `for` loop by defining an `each` function for it. When an `each` function exists that takes a value of type `T` and returns an `iterator`, you can iterate over `T` directly — the compiler calls `each` automatically:

```
struct Foo {
    data : array<int>
}

def each ( f : Foo ) : iterator<int&> {
    return each(f.data)
}
```

(continues on next page)

(continued from previous page)

```
var f = Foo(data <- [1, 2, 3])
for ( x in f ) {           // equivalent to: for ( x in each(f) )
    print("{x}\n")
}
```

This is how built-in types like ranges, arrays, and strings become iterable — each has a corresponding `each` function defined in the standard library.

Calling `delete` on an iterator will make it sequence out and free its memory:

```
var it <- each_enum(Numbers.one)
delete it                          // safe to delete

var it <- each_enum(Numbers.one)
for ( x in it ) {
    print("x = {x}\n")
}
delete it                          // its always safe to delete sequenced out iterator
```

Loops and iteration functions call the finalizer automatically.

2.20.2 builtin iterators

Table keys and values iterators can be obtained via the `keys` and `values` functions:

```
var tab <- { "one"=>1, "two"=>2 }
for ( k,v in keys(tab),values(tab) ) {  // keys(tab) is iterator<string>
    print("{k} => {v}\n")               // values(tab) is iterator<int&>
}
```

It is possible to iterate over each character of the string via the `each` function:

```
unsafe {
    for ( ch in each("hello,world!") ) {   // string iterator is iterator<int>
        print("ch = {ch}\n")
    }
}
```

It is possible to iterate over each element of an enumeration via the `each_enum` function:

```
enum Numbers {
    one
    two
    ten = 10
}

for ( x in each_enum(Numbers.one) ) {       // argument is any value from said␣
↪enumeration
    print("x = {x}\n")
}
```

2.20.3 builtin iteration functions

The `empty` function checks if an iterator is null or already sequenced out:

```
unsafe {
    var it <- each ( fixed_array(1,2,3,4) )
    for ( x in it ) {
        print("x = {x}\n")
    }
    verify(empty(it))           // iterator is sequenced out
}
```

More complicated iteration patterns may require the `next` function:

```
var x : int
while ( next(it,x) ) {       // this is semantically equivalent to the `for x in it`
    print("x = {x}\n")
}
```

Next can only operate on copyable types.

2.20.4 low level builtin iteration functions

`_builtin_iterator_first`, `_builtin_iterator_next`, and `_builtin_iterator_close` address the regular lifecycle of the iterator. A semantic equivalent of the for loop can be explicitly written using these operations:

```
var it <- each(range(0,10))
var i : int
var pi : void?
unsafe {
    pi = reinterpret<void?> ( addr(i) )
}
if ( _builtin_iterator_first(it,pi) ) {
    print("i = {i}\n")
    while ( _builtin_iterator_next(it,pi) ) {
        print("i = {i}\n")
    }
    _builtin_iterator_close(it,pi)
}
```

`_builtin_iterator_iterate` is one function to rule them all. It acts like all 3 functions above. On a non-empty iterator it starts with 'first', then proceeds to call `next` until the sequence is exhausted. Once the iterator is sequenced out, it calls `close`:

```
var it <- each(range(0,10))
var i : int
var pi : void?
unsafe {
    pi = reinterpret<void?> ( addr(i) )
}
while ( _builtin_iterator_iterate(it,pi) ) {     // this is equivalent to the example␣
↪above
    print("i = {i}\n")
}
```

2.20.5 next implementation details

The function `next` is implemented as follows:

```
def next ( it:iterator<auto(TT)>; var value : TT& ) : bool {
    static_if (!typeinfo can_copy(type<TT>)) {
        concept_assert(false, "requires type
         which can be copied")
    } static_elif (typeinfo is_ref_value(type<TT>)) {
        var pValue : TT - & ?
        unsafe {
            if ( _builtin_iterator_iterate(it, addr(pValue)) ) {
                value = *pValue
                return true
            } else {
                return false
            }
        }
    } else {
        unsafe {
            return _builtin_iterator_iterate(it, addr(value))
        }
    }
}
```

It is important to notice that builtin iteration functions accept pointers instead of references.

See also:

Arrays and *Tables* for built-in iterable containers, *Statements* for the `for` loop syntax, *Comprehensions* for comprehension-based iterators, *Generators* for creating iterators from generator functions, *Finalizers* for iterator finalization.

2.21 Comprehension

Comprehensions are concise notation constructs designed to allow sequences to be built from other sequences.

The syntax is inspired by that of a `for` loop. Comprehensions produce either a dynamic array, an iterator, or a table, depending on the style of brackets used:

- `[ for(...); expr ]` — produces a dynamic array
- `[iterator [ for(...); expr ]]` — produces an iterator
- `{ for(...); key_expr => value_expr }` — produces a table
- `{ for(...); key_expr }` — produces a key-only table (set)

An optional `where` clause can filter elements.

Examples:

```
var a1 <- [iterator for(x in range(0,10)); x]   // iterator<int>
var a2 <- [for(x in range(0,10)); x]            // array<int>
var at1 <- {for(x in range(0,10)); x}           // table<int>
var at2 <- {for(x in range(0,10)); x=>"{x}"}    // table<int;string>
```

A `where` clause acts as a filter:

```
var a3 <- [for(x in range(0,10)); x; where (x & 1) == 1]   // only odd numbers
```

Just like a for loop, comprehension can iterate over multiple sources:

```
var a4 <- [for(x,y in range(0,10),a1); x + y; where x==y] // multiple variables
```

Iterator comprehension may produce a referenced iterator:

```
var a = [1,2,3,4]
var b <- [iterator for(x in a); a]  // iterator<int&> and will point to captured copy of␣
↪the elements of a
```

Regular lambda capturing rules apply to iterator comprehensions (see *Lambdas*).

Internally array comprehension produces an `invoke` of a local block and a for loop; whereas iterator comprehension produces a generator (lambda). Array comprehensions are typically faster, but iterator comprehensions have less of a memory footprint.

See also:

Arrays for the array type produced by array comprehensions, *Tables* for the table type produced by table comprehensions, *Iterators* for the iterator type produced by iterator comprehensions, *Generators* for the generator mechanism underlying iterator comprehensions.

2.22 String Builder

Daslang provides built-in string interpolation for constructing strings from expressions at runtime. Any string literal can contain embedded expressions enclosed in curly brackets `{}`, which are evaluated, converted to text, and spliced into the resulting string.

This approach is more readable, compact, and type-safe than `printf`-like formatting.

2.22.1 Basic Usage

Embed any expression inside `{` and `}` within a string literal:

```
let name = "world"
let greeting = "Hello, {name}!"          // "Hello, world!"
let result = "1 + 2 = {1 + 2}"          // "1 + 2 = 3"
```

The expression inside `{}` can be arbitrarily complex:

```
let items : array<string>
push(items, "apple")
push(items, "banana")
print("count = {length(items)}, first = {items[0]}\n")
```

If the expression is a compile-time constant, the string builder result is computed at compile time.

2.22.2 Supported Types

Any type can appear inside `{}`, including:

- Numeric types (`int`, `uint`, `float`, `double`, `int64`, `uint64`)
- Booleans
- Strings
- Enumerations (printed as their name)
- Vectors (`float2`, `int3`, etc.)
- Structures and classes (via the Data Walker)
- Handled (extern) types, provided they implement `DataWalker`

All built-in POD types have `DataWalker` support by default.

2.22.3 Format Specifiers

A format specifier can be added after a colon `:` to control the output representation of the interpolated value:

```
let pi = 3.14159
print("pi = {pi:5.2f}\n")          // fixed-point, 5 wide, 2 decimals
```

Format specifiers follow a syntax similar to C `printf` format strings. The general form is:

```
{expression:flags width.precision type}
```

Where:

- **flags** — optional characters such as `-` (left-align), `+` (force sign), `0` (zero-pad)
- **width** — minimum field width
- **precision** — number of decimal places (for floating-point) or maximum string length
- **type — conversion character:**
  - `d` or `i` — signed decimal integer
  - `u` — unsigned decimal integer
  - `x` — hexadecimal (lowercase)
  - `X` — hexadecimal (uppercase)
  - `o` — octal
  - `f` — fixed-point decimal
  - `e` — scientific notation
  - `E` — scientific notation (uppercase)
  - `g` — general (shortest of `f` or `e`)
  - `G` — general (shortest of `f` or `E`)

Examples:

```
print("{42:08x}\n")            // "0000002a"  - 8-digit zero-padded hex
print("{42:08X}\n")            // "0000002A"  - uppercase hex
print("{3.14159:.2f}\n")       // "3.14"      - 2 decimal places
print("{-7:+d}\n")             // "-7"        - with sign
print("{255:#x}\n")            // "0xff"      - with 0x prefix
```

2.22.4 Escaping Curly Brackets

To include a literal { or } in a string, escape them with a backslash:

```
print("Use \{curly\} brackets\n")   // prints: Use {curly} brackets
```

2.22.5 Multi-line Strings

String interpolation works in multi-line (heredoc) strings as well:

```
let msg = "Line 1: {value1}
Line 2: {value2}
Line 3: {value3}"
```

2.22.6 Implementation Notes

String builder expressions are compiled as a sequence of `write` calls to an internal string buffer. This means:

- Each {} expression creates one write call, not a separate string allocation.
- The entire string builder expression produces a single temporary string.
- If all parts are compile-time constants, the result is folded into a single string constant.

2.22.7 Relationship to print

The `print` function accepts string builder strings directly:

```
print("x = {x}, y = {y}\n")
```

This is the most common use of string builders. The `print` function will output the result using the host application's output handler.

See also:

Datatypes for the built-in types supported inside {}, *Expressions* for expression syntax used in string interpolation.

2.23 Modules

Modules provide infrastructure for code reuse, as well as mechanism to expose C++ functionality to Daslang. A module is a collection of types, constants, and functions. Modules can be native to Daslang, as well as built-in.

For an overview of how `module`, `require`, and `options` fit into the overall file layout, see *Program Structure*.

To request a module, use the `require` keyword:

```
require math
require ast public
require daslib/ast_boost
```

The `public` modifier indicates that the included module is visible to everything that includes the current module.

Module names may contain `/` and `.` symbols. The project is responsible for resolving module names into file names (see *Project*).

2.23.1 Native modules

A native module is a separate Daslang file, with an optional `module` name:

```
module custom       // specifies module name
...
def foo             // defines function in module
...
```

If not specified, the module name defaults to that of the file name.

Modules can be `private` or `public`:

```
module Foo private

module Foo public
```

The default publicity of functions, structures, and enumerations is that of the module (i.e. if the module is public and a function's publicity is not specified, that function is public).

Module can be made visible to all modules in the project via the `inscope` modifier:

```
module Foo inscope
```

2.23.2 Builtin modules

Built-in modules are the way to expose C++ functionality to Daslang (see *Builtin modules*).

2.23.3 Shared modules

Shared modules are modules that are shared between compilation of multiple contexts. Typically, modules are compiled anew for each context, but when the 'shared' keyword is specified, the module gets promoted to a builtin module:

```
module Foo shared
```

That way only one instance of the module is created per compilation environment. Macros in shared modules can't expect the module to be unique, since sharing of the modules can be disabled via the code of policies.

2.23.4 Module function visibility

When calling a function, the name of the module can be specified explicitly or implicitly:

```
let s1 = sin(0.0)           // implicit, assumed math::sin
let s2 = math::sin(0.0)     // explicit, always math::sin
```

If the function does not exist in that module, a compilation error will occur. If the function is private or not directly visible, a compilation error will occur. If multiple functions match an implicit function call, a compilation error will occur.

Module names _ and __ are reserved to specify the *current module* and the *current module only*, respectively. It is particularly important for generic functions, which are always instanced as private functions in the current module:

```
module b

[generic]
def from_b_get_fun_4() {
    return  _::fun_4()      //  call `fun_4', as if it was implicitly called from b
}

[generic]
def from_b_get_fun_5() {
    return  __::fun_5()     // always b::fun_5
}
```

Specifying an empty prefix is the same as specifying no prefix.

Without the _ or __ module prefixes, overwritten functions would not be visible from generics. That is why the `:=` and `delete` operators are always replaced with `_::clone` or `_::finalize` calls.

See also:

Program structure for module declaration and `require` statements, *Constants and enumerations* for module-scoped constants, *Options* for per-module options.

2.24 Move, Copy, and Clone

Daslang has three assignment operators that control how values are transferred between variables. Understanding when to use each one is essential for writing correct code.

Operator	Name	Effect
=	Copy	Bitwise copy. Source remains unchanged.
<-	Move	Transfers ownership. Source is zeroed.
:=	Clone	Deep copy. Source remains unchanged.

2.24.1 Copy (=)

The copy operator performs a bitwise copy of the right-hand side into the left-hand side. The source value is not modified:

```
var a = 10
var b = a       // b is now 10, a is still 10
```

Copy works for all POD types (`int`, `float`, `bool`, `string`, pointers, etc.) and for structs whose fields are all copyable.

Types that manage owned resources — `array`, `table`, `lambda`, and `iterator` — cannot be copied. Attempting to copy them produces:

```
// error: this type can't be copied, use move (<-) or clone (:=) instead
```

Relaxed assign

By default, the compiler automatically promotes = to <- when:

- The right-hand side is a temporary value (struct literal, function return value, `new` expression)
- The type cannot be copied but can be moved

This means you can often write = and the compiler will do the right thing:

```
var a : array<int>
a = get_data()          // automatically becomes: a <- get_data()
```

This behavior is controlled by the `relaxed_assign` option (default `true`). Set `options relaxed_assign = false` to require explicit <- in all cases (see *Options*).

2.24.2 Move (<-)

The move operator transfers ownership of a value. After a move, the source is zeroed:

```
var a : array<int>
a |> push(1)
a |> push(2)

var b <- a              // b now owns the array, a is empty
```

Move is the primary way to transfer containers and other non-copyable types.

When to use move

Use <- when:

- You are done with the source variable and want to transfer its contents
- You are initializing a variable from a function return value
- You are passing ownership into a struct field or container

```
def make_data() : array<int> {
    var result : array<int>
    result |> push(1)
    result |> push(2)
    return <- result        // move out of the function
}

var data <- make_data()     // move into the variable
```

Return by move

Functions that return non-copyable types must use `return <-`:

```
def make_table() : table<string; int> {
    var t : table<string; int>
    t |> insert("one", 1)
    return <- t
}
```

A regular `return` would attempt to copy the value, which fails for non-copyable types.

2.24.3 Clone (:=)

The clone operator creates a deep copy of the right-hand side. Unlike = (which is a shallow bitwise copy), := recursively clones all nested containers and non-POD fields:

```
var a : array<int>
a |> push(1)
a |> push(2)

var b : array<int>
b := a                  // b is a deep copy; a is unchanged
```

After cloning, a and b are completely independent — modifying one does not affect the other.

(see *Clone* for implementation details and auto-generated clone functions).

When to use clone

Use `:=` when:

- You need to duplicate a container (array, table)
- You need an independent copy of a struct that contains non-copyable fields
- You want both the source and destination to remain valid after the operation

Clone initialization

You can clone-initialize a variable at declaration:

```
var a : array<int>
a |> push(1)

var b := a              // clone a into a new variable b
```

This expands into:

```
var b <- clone_to_move(a)
```

where `clone_to_move` creates a temporary clone and moves it into the new variable.

For POD types, clone initialization is optimized to a plain copy.

2.24.4 Type Compatibility

The following table summarizes which operators work with which types:

Type	= (copy)	<- (move)	:= (clone)
`int`, `float`, POD scalars	✓	✓	✓ (becomes copy)
`string`	✓	✓	✓
`array<T>`	×	✓	✓
`table<K;V>`	×	✓	✓
Struct (all POD fields)	✓	✓	✓ (becomes copy)
Struct (has array/table fields)	×	✓	✓
`tuple`	depends on elements	depends on elements	depends on elements
`variant`	depends on elements	depends on elements	depends on elements
Raw pointer	✓	✓	✓
`smart_ptr`	×	✓	✓
`lambda`	×	✓	×
`block`	×	×	×
`iterator`	×	✓	×

A struct, tuple, or variant is copyable/moveable/cloneable only if **all** of its fields are.

2.24.5 Variable Initialization

The three initialization forms correspond to the three operators:

```
var x = expr            // copy initialization
var x <- expr           // move initialization
var x := expr           // clone initialization
```

For local variable declarations, the compiler checks the type and reports an error if the chosen initialization mode is not supported:

```
var a = get_array()     // error if relaxed_assign is false:
                        // "local variable can only be move-initialized; use <- for that"
```

2.24.6 Struct Initialization

In struct literals, each field can use a different initialization mode:

```
struct Foo {
    name : string
    data : array<int>
}

var items : array<int>
items |> push(1)

var f = Foo(name = "hello", data <- items)
```

Here `name` is copy-initialized and `data` is move-initialized. After this, `items` is empty.

Clone initialization is also supported in struct literals:

```
var f2 = Foo(name = "hello", data := items)
```

After this, `items` still contains its original data.

2.24.7 Lambda Captures

Lambda capture lists support all three modes. The `capture` keyword introduces the capture list, with each entry specifying a mode and a variable name:

```
def make_lambda(a : int) {
    var b = 13
    return @ capture(= a, := b) (c : int) {
        debug(a)
        debug(b)
        debug(c)
    }
}
```

Here `= a` captures `a` by copy and `:= b` captures `b` by clone.

Each capture entry uses an operator prefix to specify the mode:

Shorthand	Named	Mode	Requirement
`& x`	`ref(x)`	Reference	Variable must outlive the lambda. May require `unsafe`.
`= x`	`copy(x)`	Copy	Type must be copyable.
`<- x`	`move(x)`	Move	Type must be moveable. Source is zeroed.
`:= x`	`clone(x)`	Clone	Type must be cloneable.

Multiple captures are separated by commas:

```
return @ capture(= a, <- arr, := table) () {
    // a is copied, arr is moved, table is cloned
}
```

Generators also support captures:

```
var g <- generator<int> capture(= a) {
    for (x in range(1, a)) {
        yield x
    }
    return false
}
```

(see *Lambdas*).

2.24.8 Containers

`array` and `table` types cannot be copied. This is because a bitwise copy would create two variables pointing to the same underlying memory, leading to double-free errors.

To transfer a container, use move:

```
var a : array<int>
a |> push(1)
var b <- a              // a is now empty
```

To duplicate a container, use clone:

```
var a : array<int>
a |> push(1)
var b : array<int>
b := a                  // independent deep copy
```

Clone of `array<T>` resizes the destination and clones each element. Clone of `table<K;V>` clears the destination and re-inserts each key-value pair.

2.24.9 Classes

Copying or moving class values requires `unsafe`:

```
class Foo {
    x : int
}

unsafe {
    var a = new Foo(x=1)
    var b = *a              // copy requires unsafe
}
```

2.24.10 Custom Clone

You can define a custom clone function for any type. If a custom clone exists, it is called by the `:=` operator regardless of whether the type is natively cloneable:

```
struct Connection {
    id : int
    socket : int
}

def clone(var dest : Connection; src : Connection) {
    dest.id = src.id
    dest.socket = open_new_socket()   // custom logic instead of bitwise copy
    print("cloned connection {src.id}\n")
}

var a = Connection(id=1, socket=42)
var b : Connection
b := a                            // calls custom clone
```

(see *Clone* for auto-generated clone functions for structs, tuples, variants, arrays, and tables).

2.24.11 Quick Reference

Here is a complete example showing all three operators:

```
options gen2

def make_data() : array<int> {
    var result : array<int>
    result |> push(1)
    result |> push(2)
    result |> push(3)
    return <- result
}

[export]
def main {
```

(continues on next page)

(continued from previous page)

```
    // Copy (scalars)
    var a = 10
    var b = a
    print("copy: a={a} b={b}\n")

    // Move (containers)
    var data <- make_data()
    print("data = {data}\n")

    var moved <- data
    print("moved = {moved}\n")
    print("data after move = {data}\n")

    // Clone (deep copy)
    var cloned : array<int>
    cloned := moved
    cloned |> push(4)
    print("cloned = {cloned}\n")
    print("moved = {moved}\n")
}
```

Expected output:

```
copy: a=10 b=10
data = [[ 1; 2; 3]]
moved = [[ 1; 2; 3]]
data after move = [[]]
cloned = [[ 1; 2; 3; 4]]
moved = [[ 1; 2; 3]]
```

I want to transfer ownership (source becomes empty):
Use `<-` (move)

I want an independent deep copy (both remain valid):
Use `:=` (clone)

I want a simple value copy (POD types):
Use `=` (copy)

I'm returning a container from a function:
Use `return <-`

I'm initializing a variable from a function call:
Use `var x <- func()` or rely on relaxed assign with `var x = func()`

The compiler says "this type can't be copied":
The type contains arrays, tables, or other non-copyable fields. Use `<-` to move or `:=` to clone.

See also:

Clone for custom clone operator implementation, *Finalizers* for delete-after-move semantics, *Arrays* and *Tables* for non-copyable container types, *Structs* and *Classes* for struct and class copy/move rules, *Temporary types* for temporary-type clone rules.

2.25 Clone

Clone is designed to create a deep copy of the data.

For an overview of when to use copy (=), move (<-), and clone (:=), see *Move, Copy, and Clone*.

Cloning is invoked via the clone operator `:=`:

```
a := b
```

Cloning can be also invoked via the clone initializer in a variable declaration:

```
var x := y
```

This in turn expands into `clone_to_move`:

```
var x <- clone_to_move(y)
```

(see *clone_to_move*).

2.25.1 Cloning rules and implementation details

Cloning obeys the following rules.

Certain types like blocks, lambdas, and iterators can't be cloned at all.

However, if a custom clone function exists, it is immediately called regardless of the type's cloneability:

```
struct Foo {
    a : int
}

def clone ( var x : Foo; y : Foo ) {
    x.a = y.a
    print("cloned\n")
}

var l = Foo(a=1)
var cl : Foo
cl := l                 // invokes clone(cl,l)
```

Cloning is typically allowed between regular and temporary types (see *Temporary types*).

POD types are copied instead of cloned:

```
var a,b : int
var c,d : int[10]
a := b
c := d
```

This expands to:

```
a = b
c = d
```

Handled types provide their own clone functionality via `canClone`, `simulateClone`, and appropriate `das_clone` C++ infrastructure (see *Handles*).

For static arrays, the `clone_dim` generic is called, and for dynamic arrays, the `clone` generic is called. Those in turn clone each of the array elements:

```
struct Foo {
    a : array<int>
    b : int
}

var a, b : array<Foo>
b := a
var c, d : Foo[10]
c := d
```

This expands to:

```
def builtin`clone ( var a:array<Foo aka TT> explicit; b:array<Foo> const ) {
    resize(a,length(b))
    for ( aV,bV in a,b ) {
        aV := bV
    }
}

def builtin`clone_dim ( var a:Foo[10] explicit; b:Foo const[10] implicit explicit ) {
    for ( aV,bV in a,b ) {
        aV := bV
    }
}
```

For tables, the `clone` generic is called, which in turn clones its values:

```
var a, b : table<string;Foo>
b := a
```

This expands to:

```
def builtin`clone ( var a:table<string aka KT;Foo aka VT> explicit; b:table<string;Foo>␣
↪const ) {
    clear(a)
    for ( k,v in keys(b),values(b) ) {
        a[k] := v
    }
}
```

For structures, the default `clone` function is generated, in which each element is cloned:

```
struct Foo {
    a : array<int>
    b : int
}
```

This expands to:

```
def clone ( var a:Foo explicit; b:Foo const ) {
    a.a := b.a
    a.b = b.b   // note copy instead of clone
}
```

For tuples, each individual element is cloned:

```
var a, b : tuple<int;array<int>;string>
b := a
```

This expands to:

```
def clone ( var dest:tuple<int;array<int>;string> -const; src:tuple<int;array<int>;
↪string> const -const ) {
    dest._0 = src._0
    dest._1 := src._1
    dest._2 = src._2
}
```

For variants, only the currently active element is cloned:

```
var a, b : variant<i:int;a:array<int>;s:string>
b := a
```

This expands to:

```
def clone ( var dest:variant<i:int;a:array<int>;s:string> -const; src:variant<i:int;
↪a:array<int>;s:string> const -const ) {
    if ( src is i ) {
        set_variant_index(dest,0)
        dest.i = src.i
    } elif ( src is a ) {
        set_variant_index(dest,1)
        dest.a := src.a
    } elif ( src is s ) {
        set_variant_index(dest,2)
        dest.s = src.s
    }
}
```

2.25.2 clone_to_move implementation

`clone_to_move` is implemented via regular generics as part of the builtin module:

```
def clone_to_move(clone_src:auto(TT)) : TT -const {
    var clone_dest : TT
    clone_dest := clone_src
    return <- clone_dest
}
```

Note that for non-cloneable types, Daslang will not promote `:=` initialize into `clone_to_move`.

See also:

Move, copy, and clone for move, copy, and assignment rules, *Finalizers* for delete and finalization semantics, *Structs* and *Variants* for custom clone expansion.

2.26 Finalizer

Finalizers are special functions called in exactly two cases:

`delete` is called explicitly on a data type:

```
var f <- [1,2,3,4]
delete f
```

Lambda based iterator or generator is sequenced out:

```
var src <- [1,2,3,4]
var gen <- generator<int&> capture (move(src)) ( $ {
    for ( w in src ) {
        yield w
    }
    return false
})
for ( t in gen ) {
    print("t = {t}\n")
}
// finalizer is called on captured version of src
```

By default finalizers are called recursively on subtypes.

If the memory model allows deallocation, standard finalizers also free the memory:

```
options persistent_heap = true

var src <- [1,2,3,4]
delete src                      // memory of src will be freed here
```

Custom finalizers can be defined for any type by overriding the `finalize` function. Generic custom finalizers are also allowed:

```
struct Foo {
    a : int
}
def finalize ( var foo : Foo ) {
    print("we kill foo\n")
}
var f = Foo(a = 5)
delete f                    // prints 'we kill foo' here
```

2.26.1 Rules and implementation details

Finalizers obey the following rules:

If a custom `finalize` is available, it is called instead of default one.

Pointer finalizers expand to calling `finalize` on the dereferenced pointer, and then calling the native memory finalizer on the result:

```
var pf = new Foo
unsafe {
    delete pf
}
```

This expands to:

```
def finalize ( var __this:Foo?& explicit -const ) {
    if ( __this != null ) {
        _::finalize(deref(__this))
        delete /*native*/ __this
        __this = null
    }
}
```

Static arrays call `finalize_dim` generically, which finalizes all its values:

```
var f : Foo[5]
delete f
```

This expands to:

```
def builtin`finalize_dim ( var a:Foo aka TT[5] explicit ) {
    for ( aV in a ) {
        _::finalize(aV)
    }
}
```

Dynamic arrays call `finalize` generically, which finalizes all its values:

```
var f : array<Foo>
delete f
```

This expands to:

```
def builtin`finalize ( var a:array<Foo aka TT> explicit ) {
    for ( aV in a ) {
        _::finalize(aV)
    }
    __builtin_array_free(a,4,__context__)
}
```

Tables call `finalize` generically, which finalizes all its values, but not its keys:

```
var f : table<string;Foo>
delete f
```

This expands to:

```
def builtin`finalize ( var a:table<string aka TK;Foo aka TV> explicit ) {
    for ( aV in values(a) ) {
        _::finalize(aV)
    }
    __builtin_table_free(a,8,4,__context__)
}
```

Custom finalizers are generated for structures. Fields annotated as @do_not_delete are ignored. `memzero` clears structure memory at the end:

```
struct Goo {
    a : Foo
    @do_not_delete b : array<int>
}

var g <- default<Goo>
delete g
```

This expands to:

```
def finalize ( var __this:Goo explicit ) {
    _::finalize(__this.a)
    __::builtin`finalize(__this.b)
    memzero(__this)
}
```

Tuples behave similar to structures. There is no way to ignore individual fields:

```
var t : tuple<Foo; int>
delete t
```

This expands to:

```
def finalize ( var __this:tuple<Foo;int> explicit -const ) {
    _::finalize(__this._0)
    memzero(__this)
}
```

Variants behave similarly to tuples. Only the currently active variant is finalized:

```
var t : variant<f:Foo; i:int; ai:array<int>>
delete t
```

This expands to:

```
def finalize ( var __this:variant<f:Foo;i:int;ai:array<int>> explicit -const ) {
    if ( __this is f ) {
        _::finalize(__this.f)
    } else if (__this is ai) {
        __::builtin`finalize(__this.ai)
    }
    memzero(__this)
}
```

Lambdas and generators have their capture structure finalized. Lambdas can have custom finalizers defined as well (see *Lambdas*).

Classes can define custom finalizers inside the class body (see *Classes*).

See also:

Structs for struct finalizer expansion, *Tuples* and *Variants* for composite type finalization, *Arrays* and *Tables* for container finalization, *Move, copy, and clone* for delete-after-move semantics, *Options* for `persistent_heap` and memory deallocation policies.

2.27 Temporary types

Temporary types are designed to address lifetime issues of data, which are exposed to Daslang directly from C++.

Let's review the following C++ example:

```
void peek_das_string(const string & str, const TBlock<void,TTemporary<const char *>> &␣
↪block, Context * context) {
    vec4f args[1];
    args[0] = cast<const char *>::from(str.c_str());
    context->invoke(block, args, nullptr);
}
```

The C++ function here exposes a pointer a to c-string, internal to std::string. From Daslang's perspective, the declaration of the function looks like this:

```
def peek ( str : das_string; blk : block<(arg:string#):void> )
```

Where string# is a temporary version of a Daslang string type.

The key property of temporary types is that they cannot escape the scope of the block they are passed to.

Temporary values enforce this through the following rules.

Temporary values can't be copied or moved:

```
def sample ( var t : das_string ) {
    var s : string
    peek(t) $ ( boo : string# ) {
        s = boo // error, can't copy temporary value
    }
}
```

Temporary values can't be returned or passed to functions, which require regular values:

```
def accept_string(s:string) {
    print("s={s}\n")
}

def sample ( var t : das_string ) {
    peek(t) $ ( boo : string# ) {
        accept_string(boo) // error
    }
}
```

This causes the following error:

```
30304: no matching functions or generics accept_string ( string const&# )
candidate function:
        accept_string ( s : string const ) : void
                invalid argument s. expecting string const, passing string const&#
```

Values need to be marked as `implicit` to accept both temporary and regular values. These functions implicitly promise that the data will not be cached (copied, moved) in any form:

```
def accept_any_string(s:string implicit) {
    print("s={s}\n")
}

def sample ( var t : das_string ) {
    peek(t) $ ( boo : string# ) {
        accept_any_string(boo)
    }
}
```

Temporary values can and are intended to be cloned:

```
def sample ( var t : das_string ) {
    peek(t) $ ( boo : string# ) {
        var boo_clone : string := boo
        accept_string(boo_clone)
    }
}
```

Returning a temporary value is an unsafe operation.

A pointer to the temporary value can be received for the corresponding scope via the `safe_addr` macro:

```
require daslib/safe_addr

def foo {
    var a = 13
    ...
    var b = safe_addr(a)    // b is int?#, and this operation does not require unsafe
    ...
}
```

See also:

Unsafe for `implicit` keyword usage and returning temporary values, *Clone* for cloning temporary values into regular ones, *Blocks* for temporary scope and block lifetime, *Functions* for `implicit` in function signatures.

2.28 Unsafe

The `unsafe` keyword denotes unsafe contents, which is required for operations, but could potentially crash the application:

```
unsafe {
    let px = addr(x)
}
```

Expressions (and subexpressions) can also be unsafe:

```
let px = unsafe(addr(x))
```

The `unsafe` keyword is followed by a block that can include such operations. Nested unsafe sections are allowed. `unsafe` is not inherited in lambdas, generators, or local functions, but it is inherited in local blocks.

Individual expressions can cause a `CompilationError::unsafe` error, unless they are part of the unsafe section. Additionally, macros can explicitly set the `ExprGenFlags::alwaysSafe` flag.

The address of expression is unsafe:

```
unsafe {
    let a : int
    let pa = addr(a)
    return pa                                   // accessing *pa can potentially corrupt␣
↪stack
}
```

Lambdas or generators require unsafe sections for the implicit capture by move or by reference:

```
var a : array<int>
unsafe {
    var counter <- @ (extra:int) : int {
        return a[0] + extra                     // a is implicitly moved
    }
}
```

Deleting any pointer requires an unsafe section:

```
var p = new Foo()
var q = p
unsafe {
    delete p                                    // accessing q can potentially corrupt memory
}
```

Upcast and reinterpret cast require an unsafe section:

```
unsafe {
    return reinterpret<void?> 13                // reinterpret can create unsafe pointers
}
```

Indexing into a pointer is unsafe:

```
unsafe {
    var p = new Foo()
```

(continues on next page)

(continued from previous page)

```
    return p[13]                            // accessing out of bounds pointer can␣
↪potentially corrupt memory
}
```

A safe index is unsafe when not followed by the null coalescing operator:

```
var a = { 13 => 12 }
unsafe {
    var t = a?[13] ?? 1234                  // safe
    return a?[13]                           // unsafe; safe index is a form of 'addr'␣
↪operation
                                            // it can create pointers to temporary␣
↪objects
}
```

Variant `?as` on local variables is unsafe when not followed by the null coalescing operator:

```
unsafe {
    return a ?as Bar                        // safe as is a form of 'addr' operation
}
```

Variant `.?field` is unsafe when not followed by the null coalescing operator:

```
unsafe {
    return a?.Bar                           // safe navigation of a variant is a form of
↪'addr' operation
}
```

Variant `.field` is unsafe:

```
unsafe {
    return a.Bar                            // this is potentially a reinterpret cast
}
```

Certain functions and operators are inherently unsafe or marked unsafe via the [unsafe_operation] annotation:

```
unsafe {
    var a : int?
    a += 13                                 // pointer arithmetic can create invalid␣
↪pointers
    var boo : int[13]
    var it = each(boo)                      // each() of array is unsafe, for it does␣
↪not capture
}
```

Moving from a smart pointer value requires unsafe, unless that value is the ‘new’ operator:

```
unsafe {
    var a <- new TestObjectSmart()          // safe, its explicitly new
    var b <- someSmartFunction()            // unsafe since lifetime is not obvious
    b <- a                                  // safe, values are not lost
}
```

Moving or copying classes is unsafe:

```
def foo ( var b : TestClass ) {
    unsafe {
        var a : TestClass
        a <- b                              // potentially moving from derived class
    }
}
```

Local class variables are unsafe:

```
unsafe {
    var g = Goo()                           // potential lifetime issues
}
```

2.28.1 implicit

`implicit` keyword is used to specify that type can be either temporary or regular type, and will be treated as defined. For example:

```
def foo ( a : Foo implicit )    // a will be treated as Foo, but will also accept Foo#␣
↪as argument
def foo ( a : Foo# implicit )   // a will be treated as Foo#, but will also accept Foo␣
↪as argument
```

Unfortunately implicit conversions like this are unsafe, so `implicit` is unsafe by definition.

2.28.2 other cases

There are several other cases where `unsafe` is required, but not explicitly mentioned in the documentation. They are typically controlled via CodeOfPolicies or appropriate option:

```
options unsafe_table_lookup = false // makes table indexing safe. refers to␣
↪CodeOfPolicies::unsafe_table_lookup

var tab <- { 1=>"one", 2=>"two" }
tab[3] = "three"        // this is unsafe, since it can create a pointer to a temporary␣
↪object
```

See also:

Lambdas for capture-by-reference and move requiring unsafe, *Classes* for local class variables being unsafe, *Expressions* for `addr`, `reinterpret`, and `upcast` operators, *Temporary types* for `implicit` and temporary type qualifiers, *Tables* for unsafe table lookup, *Options* for `unsafe_table_lookup` and related policies.

2.29 Generic Programming

Daslang allows you to omit types in statements, functions, and function declarations, making the code similar in style to dynamically typed languages such as Python or Lua. Said functions are *instantiated* for specific types of arguments on the first call.

There are also ways to inspect the types of the provided arguments, in order to change the behavior of a function, or to provide meaningful errors during the compilation phase.

Unlike C++ with its SFINAE, you can use common conditionals (if) in order to change the instance of the function depending on the type info of its arguments. Consider the following example:

```
def setSomeField(var obj; val) {
    if ( typeinfo has_field<someField>(obj) ) {
        obj.someField = val
    }
}
```

This function sets `someField` in the provided argument *if* it is a struct with a `someField` member.

We can do even more. For example:

```
def setSomeField(var obj; val: auto(valT)) {
    if ( typeinfo has_field<someField>(obj) ) {
        if ( typeinfo typename(obj.someField) == typeinfo typename(type<valT -const>) ) {
            obj.someField = val
        }
    }
}
```

This function sets `someField` in the provided argument *if* it is a struct with a `someField` member, and only if `someField` is of the same type as `val`!

2.29.1 typeinfo

The `typeinfo` operator provides compile-time type reflection. It is the primary mechanism for inspecting types in generic functions.

All `typeinfo` traits can operate on either an expression or a `type<T>` argument:

```
typeinfo(sizeof type<float3>)       // 12
typeinfo(typename my_variable)      // type name of the variable
```

Name and String Traits

- `typeinfo(typename expr)` — human-readable type name
- `typeinfo(fulltypename expr)` — full type name with contracts (`const`, `&`, etc.)
- `typeinfo(stripped_typename expr)` — type name without `const`/`temp`/`ref` decorators
- `typeinfo(undecorated_typename expr)` — type name without module prefix
- `typeinfo(modulename expr)` — module name of the type
- `typeinfo(struct_name expr)` — structure name (for struct/class types)
- `typeinfo(struct_modulename expr)` — module name of the structure

Size and Layout Traits

- `typeinfo(sizeof expr)` — size of the type in bytes
- `typeinfo(alignof expr)` — alignment of the type
- `typeinfo(dim expr)` — size of the first dimension (fixed-size arrays)
- `typeinfo(offsetof<field> expr)` — byte offset of a field in a struct
- `typeinfo(vector_dim expr)` — dimension of a vector type (e.g. 3 for `float3`)

Boolean Type-Query Traits

- `typeinfo(is_pod expr)` — true if plain-old-data
- `typeinfo(is_raw expr)` — true if raw data (can be memcpy'd)
- `typeinfo(is_struct expr)` — true if structure type
- `typeinfo(is_class expr)` — true if class
- `typeinfo(is_handle expr)` — true if handled (native-bound) type
- `typeinfo(is_ref expr)` — true if passed/stored by reference
- `typeinfo(is_ref_type expr)` — true if reference type by nature (array, table, etc.)
- `typeinfo(is_ref_value expr)` — true if has explicit `ref` qualifier
- `typeinfo(is_const expr)` — true if const-qualified
- `typeinfo(is_temp expr)` — true if temporary-qualified
- `typeinfo(is_temp_type expr)` — true if temporary type
- `typeinfo(is_pointer expr)` — true if pointer type
- `typeinfo(is_smart_ptr expr)` — true if smart pointer
- `typeinfo(is_void expr)` — true if void
- `typeinfo(is_void_pointer expr)` — true if void pointer
- `typeinfo(is_string expr)` — true if string type
- `typeinfo(is_numeric expr)` — true if numeric type
- `typeinfo(is_numeric_comparable expr)` — true if numeric-comparable
- `typeinfo(is_vector expr)` — true if vector type (`float2`/`int3`/etc.)
- `typeinfo(is_any_vector expr)` — true if handled vector-template type
- `typeinfo(is_array expr)` — true if `array<T>`
- `typeinfo(is_table expr)` — true if `table<K;V>`
- `typeinfo(is_dim expr)` — true if has any dimension (fixed-size array)
- `typeinfo(is_enum expr)` — true if enumeration
- `typeinfo(is_bitfield expr)` — true if bitfield
- `typeinfo(is_tuple expr)` — true if tuple
- `typeinfo(is_variant expr)` — true if variant
- `typeinfo(is_function expr)` — true if function type
- `typeinfo(is_lambda expr)` — true if lambda type

- `typeinfo(is_iterator expr)` — true if iterator type
- `typeinfo(is_iterable expr)` — true if can be iterated with `for`
- `typeinfo(is_local expr)` — true if local type
- `typeinfo(is_workhorse expr)` — true if workhorse type

Capability Traits

- `typeinfo(can_copy expr)` — true if the type can be copied
- `typeinfo(can_move expr)` — true if the type can be moved
- `typeinfo(can_clone expr)` — true if the type can be cloned
- `typeinfo(can_clone_from_const expr)` — true if can be cloned from a const source
- `typeinfo(can_new expr)` — true if can be heap-allocated with `new`
- `typeinfo(can_delete expr)` — true if can be deleted
- `typeinfo(can_delete_ptr expr)` — true if pointer can be deleted
- `typeinfo(can_be_placed_in_container expr)` — true if valid for arrays/tables
- `typeinfo(need_delete expr)` — true if requires explicit deletion
- `typeinfo(need_inscope expr)` — true if needs `inscope` lifetime management
- `typeinfo(need_lock_check expr)` — true if needs lock checking
- `typeinfo(has_nontrivial_ctor expr)` — true if has non-trivial constructor
- `typeinfo(has_nontrivial_dtor expr)` — true if has non-trivial destructor
- `typeinfo(has_nontrivial_copy expr)` — true if has non-trivial copy semantics
- `typeinfo(is_unsafe_when_uninitialized expr)` — true if unsafe when uninitialized

Field and Annotation Traits (see also *Annotations*)

- `typeinfo(has_field<name> expr)` — true if the struct/handle has a field named `name`
- `typeinfo(safe_has_field<name> expr)` — same as above, but returns false instead of error
- `typeinfo(has_annotation<name> expr)` — true if the struct has annotation `name`
- `typeinfo(has_annotation_argument<name> expr)` — true if annotation has argument `name`
- `typeinfo(safe_has_annotation_argument<name> expr)` — returns false instead of error
- `typeinfo(annotation_argument<name> expr)` — returns the value of an annotation argument
- `typeinfo(variant_index<name> expr)` — returns the index of a variant field
- `typeinfo(safe_variant_index<name> expr)` — returns -1 instead of error

Existence Checks

- `typeinfo(builtin_function_exists expr)` — true if a `@@function` exists
- `typeinfo(builtin_annotation_exists expr)` — true if an annotation type exists
- `typeinfo(builtin_module_exists expr)` — true if a module is loaded
- `typeinfo(is_argument expr)` — true if the expression is a function argument
- `typeinfo(mangled_name expr)` — returns the mangled name of a `@@function`

User-Defined Traits

Any trait name not in the list above is dispatched to the TypeInfoMacro system, allowing user-defined typeinfo extensions (e.g., `ast_typedecl`).

2.29.2 auto and auto(named)

Instead of omitting the type name in a generic, it is possible to use an explicit `auto` type or `auto(name)` to type it:

```
def fn(a: auto): auto {
    return a
}
```

or

```
def fn(a: auto(some_name)): some_name {
    return a
}
```

This is the same as:

```
def fn(a) {
    return a
}
```

This is very helpful if the function accepts numerous arguments, and some of them have to be of the same type:

```
def fn(a, b) { // a and b can be of different types
    return a + b
}
```

This is not the same as:

```
def fn(a, b: auto) { // a and b are one type
    return a + b
}
```

Also, consider the following:

```
def set0(a, b; index: int) { // a is only supposed to be of array type, of same type as b
    return a[index] = b
}
```

If you call this function with an array of floats and an int, you would get a not-so-obvious compiler error message:

```
def set0(a: array<auto(some)>; b: some; index: int) { // a is of array type, of same␣
↪type as b
    return a[index] = b
}
```

Usage of named `auto` with `typeinfo`

```
def fn(a: auto(some)) {
    print(typeinfo typename(type<some>))
```

(continues on next page)

(continued from previous page)

```
}

fn(1) // print "const int"
```

You can also modify the type with delete syntax:

```
def fn(a: auto(some)) {
    print(typeinfo typename(type<some -const>))
}

fn(1) // print "int"
```

2.29.3 type contracts and type operations

Generic function arguments, result, and inferred type aliases can be operated on during the inference.

`const` specifies, that constant and regular expressions will be matched:

```
def foo ( a : Foo const )   // accepts Foo and Foo const
```

`==const` specifies, that const of the expression has to match const of the argument:

```
def foo ( a : Foo const ==const )   // accepts Foo const only
def foo ( var a : Foo ==const )     // accepts Foo only
```

`-const` will remove const from the matching type:

```
def foo ( a : array<auto -const> )  // matches any array, with non-const elements
```

`#` specifies that only temporary types are accepted:

```
def foo ( a : Foo# )    // accepts Foo# only
```

`-#` will remove temporary type from the matching type:

```
def foo ( a : auto(TT) ) {      // accepts any type
    var temp : TT -# := a       // TT -# is now a regular type, and when `a` is temporary,
↪ it can clone it into `temp`
}
```

`&` specifies that argument is passed by reference:

```
def foo ( a : auto& )           // accepts any type, passed by reference
```

`==&` specifies that reference of the expression has to match reference of the argument:

```
def foo ( a : auto& ==& )   // accepts any type, passed by reference (for example␣
↪variable i, even if its integer)
def foo ( a : auto ==& )    // accepts any type, passed by value     (for example value␣
↪3)
```

`-&` will remove reference from the matching type:

```
def foo ( a : auto(TT)& ) {     // accepts any type, passed by reference
    var temp : TT -& = a        // TT -& is not a local reference
}
```

[] specifies that the argument is a static array of arbitrary dimension:

```
def foo ( a : auto[] )          // accepts static array of any type of any size
```

-[] will remove static array dimension from the matching type:

```
def take_dim( a : auto(TT) ) {
    var temp : TT -[]           // temp is type of element of a
}
// if a is int[10] temp is int
// if a is int[10][20][30] temp is still int
```

implicit specifies that both temporary and regular types can be matched, but the type will be treated as specified. implicit is _UNSAFE_:

```
def foo ( a : Foo implicit )    // accepts Foo and Foo#, a will be treated as Foo
def foo ( a : Foo# implicit )   // accepts Foo and Foo#, a will be treated as Foo#
```

explicit specifies that LSP will not be applied, and only exact type match will be accepted:

```
def foo ( a : Foo )             // accepts Foo and any type that is inherited from Foo␣
↪directly or indirectly
def foo ( a : Foo explicit )    // accepts Foo only
```

2.29.4 options

Multiple options can be specified as a function argument:

```
def foo ( a : int | float )   // accepts int or float
```

Optional types always make function generic.

Generic options will be matched in the order listed:

```
def foo ( a : Bar explicit | Foo )   // first will try to match exactly Bar, than␣
↪anything else inherited from Foo
```

|# shortcat matches previous type, with temporary flipped:

```
def foo ( a : Foo |# )   // accepts Foo and Foo# in that order
def foo ( a : Foo# |# )  // accepts Foo# and Foo in that order
```

2.29.5 typedecl

Consider the following example:

```
struct A {
    id : string
}
struct B {
    id : int
}
def get_table_from_id(t : auto(T)) {
    var tab : table<typedecl(t.id); T>  // NOTE typedecl
    return <- tab
}

[export]
def main {
    var a : A
    var b : B
    var aTable <- get_table_from_id(a)
    var bTable <- get_table_from_id(b)
    print("{typeinfo typename(aTable)}\n")
    print("{typeinfo typename(bTable)}\n")
}
```

Expected output:

```
table<string const;A const>
table<int const;B const>
```

Here table is created with a key type of `id` field of the provided struct. This feature allows to create types based on the provided expression type.

2.29.6 generic tuples and type<> expressions

Consider the following example:

```
tuple Handle {
    h : auto(HandleType)
    i : int
}

def make_handle ( t : auto(HandleType) ) : Handle {
    var h : type<Handle> // NOTE type<Handle>
    return h
}

def take_handle ( h : Handle ) {
    print("count = {h.i} of type {typeinfo typename(type<HandleType>)}\n")
}

[export]
def main {
```

(continues on next page)

(continued from previous page)

```
    let h = make_handle(10)
    take_handle(h)
}
```

Expected output:

```
count = 0 of type int const
```

In the function make_handle, the type of the variable h is created with the type<> expression. type<> is inferred in context (this time based on a function argument). This feature allows to create types based on the provided expression type.

Generic function take_handle takes any Handle type, but only Handle type tuple.

This carries some similarity to the C++ template system, but is a bit more limited due to tuples being weak types.

Module prefixes in generics

Generic functions are always instanced as private functions in the *calling* module. This means that unqualified function calls inside a generic resolve using the defining module's visible symbols — **not** the caller's.

Three prefixes control how names are resolved inside a generic:

Prefix	Resolution
(none)	Resolved in the module where the generic is **defined** — the caller's overloads are **not** visible.
`_::`	Resolved as if the call were made implicitly in the **current module** (the one that instances the generic) — the caller's overloads **are** visible.
`__::`	Resolved strictly in the module where the generic is **defined** — only that module's own symbols, nothing imported.

This distinction matters whenever a library generic should dispatch to user-provided overloads. For example:

```
// --- module "serializer" ---
module serializer

[generic]
def save(val) {
    _::write(val)       // resolves in the caller's module
}

// --- user code ---
require serializer

struct Color { r : float; g : float; b : float }

def write(c : Color) {       // user overload
    print("{c.r},{c.g},{c.b}")
}

[export]
def main() {
```

(continues on next page)

(continued from previous page)

```
    save(Color(r=1.0))      // calls user's `write(Color)` via _::
}
```

If `save` called plain `write(val)` instead of `_::write(val)`, the user's overload would not be found — the call would resolve in the `serializer` module's scope, where no `write(Color)` exists.

This is why the built-in `:=` and `delete` operators are always emitted as `_::clone` and `_::finalize` — so that user-defined `clone` and `finalize` overloads are picked up when generics are instanced in user code.

See also:

Modules for full details on `_::` and `__::` prefixes, *Functions* for regular (non-generic) function declarations, *Datatypes* for type traits and built-in types, *Temporary types* for `#/-#` temporary qualifiers, *Aliases* for `auto(Name)` aliases, *Unsafe* for the `implicit` keyword.

2.30 Pattern matching

Pattern matching allows you to compare a value against a set of structural patterns, extracting specific fields when a pattern matches. In Daslang, pattern matching is implemented via macros in the `daslib/match` module.

2.30.1 Enumeration Matching

You can match on enumeration values using the `match` keyword. Each `if` clause represents a pattern to test. The `_` pattern is a catch-all that matches anything not covered by previous cases:

```
enum Color {
    Black
    Red
    Green
    Blue
}

def enum_match (color:Color) {
    match ( color ) {
        if ( Color Black ) {
            return 0
        }
        if ( Color Red ) {
            return 1
        }
        if ( _ ) {
            return -1
        }
    }
}
```

2.30.2 Matching Variants

Variants can be matched using the `as` keyword to test and bind to a specific case:

```
variant IF {
    i : int
    f : float
}

def variant_as_match (v:IF) {
    match ( v ) {
        if ( _ as i ) {
            return "int"
        }
        if ( _ as f ) {
            return "float"
        }
        if ( _ ) {
            return "anything"
        }
    }
}
```

Variants can also be matched using constructor syntax:

```
def variant_match (v : IF) {
    match ( v ) {
        if ( IF(i=$v(i)) ) {
            return 1
        }
        if ( IF(f=$v(f)) ) {
            return 2
        }
        if ( _ ) {
            return 0
        }
    }
}
```

Here `$v(i)` declares a variable `i` that is bound to the matched variant field.

2.30.3 Declaring Variables in Patterns

The `$v(name)` syntax declares a new variable and binds it to the matched value. This works in any pattern, not just variant matching:

```
variant IF {
    i : int
    f : float
}

def variant_as_match (v:IF) {
    match ( v ) {
```

(continues on next page)

(continued from previous page)

```
        if ( $v(as_int) as i ) {
            return as_int
        }
        if ( $v(as_float) as f ) {
            return as_float
        }
        if ( _ ) {
            return None
        }
    }
}
```

2.30.4 Matching Structs

Structs can be matched by specifying field values or binding fields to variables:

```
struct Foo {
    a : int
}

def struct_match (f:Foo) {
    match ( f ) {
        if ( Foo(a=13) ) {
            return 0
        }
        if ( Foo(a=$v(anyA)) ) {
            return anyA
        }
    }
}
```

The first case matches only when a is 13. The second case matches any Foo and binds a to the variable anyA.

2.30.5 Using Guards

Guards are additional conditions that must be satisfied for a match to succeed. They are specified with && after the pattern:

```
struct AB {
    a, b : int
}

def guards_match (ab:AB) {
    match ( ab ) {
        if ( AB(a=$v(a), b=$v(b)) && (b > a) ) {
            return "{b} > {a}"
        }
        if ( AB(a=$v(a), b=$v(b)) ) {
            return "{b} <= {a}"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.30.6 Tuple Matching

Tuples are matched using value or wildcard patterns. The `...` pattern matches any number of elements:

```
def tuple_match ( A : tuple<int;float;string> ) {
    match ( A ) {
        if (1,_,"3") {
            return 1
        }
        if (13,...) {       // starts with 13
            return 2
        }
        if (...,"13") {     // ends with "13"
            return 3
        }
        if (2,...,"2") {    // starts with 2, ends with "2"
            return 4
        }
        if ( _ ) {
            return 0
        }
    }
}
```

The _ matches any single element; `...` matches zero or more elements.

2.30.7 Matching Static Arrays

Static arrays use the `fixed_array` keyword and support the same wildcard and guard patterns:

```
def static_array_match ( A : int[3] ) {
    match ( A ) {
        if ( fixed_array($v(a),$v(b),$v(c)) && (a+b+c)==6 ) { // total of 3 elements,␣
↪sum is 6
            return 1
        }
        if ( fixed_array(0,...) ) {     // starts with 0
            return 0
        }
        if ( fixed_array(..,13) ) {   // ends with 13
            return 2
        }
        if ( fixed_array(12,...,12) ) {     // starts and ends with 12
            return 3
        }
        if ( _ ) {
            return -1
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

2.30.8 Dynamic Array Matching

Dynamic arrays use bracket syntax and support the same patterns as tuples and static arrays. The number of explicit elements in the pattern is checked against the array length:

```
def dynamic_array_match ( A : array<int> ) {
    match ( A ) {
        if ( [$v(a),$v(b),$v(c)] && (a+b+c)==6 ) { // total of 3 elements, sum is 6
            return 1
        }
        if ( [0,0,0,...] ) {    // first 3 are 0
            return 0
        }
        if ( [...,1,2] ) {      // ends with 1,2
            return 2
        }
        if ( [0,1;...,2,3] ) {    // starts with 0,1, ends with 2,3
            return 3
        }
        if ( _ ) {
            return -1
        }
    }
}
```

2.30.9 Match Expressions

The `match_expr` pattern matches when an expression involving previously declared variables equals the value. This is useful for expressing relationships between elements:

```
def ascending_array_match ( A : int[3] ) {
    match ( A ) {
        if ( fixed_array($v(x),match_expr(x+1),match_expr(x+2)) ) {
            return true
        }
        if ( _ ) {
            return false
        }
    }
}
```

Here the first element is bound to `x`, and the second and third elements must equal `x+1` and `x+2` respectively.

2.30.10 Matching with ||

The || operator matches either of the provided patterns. Both sides must declare the same variables:

```
struct Bar {
    a : int
    b : float
}

def or_match ( B:Bar ) {
    match ( B ) {
        if ( Bar(a=1,b=$v(b)) || Bar(a=2,b=$v(b)) ) {
            return b
        }
        if ( _ ) {
            return 0.0
        }
    }
}
```

2.30.11 [match_as_is] Structure Annotation

The [match_as_is] annotation enables pattern matching for structures of different types, provided the necessary is and as operators have been implemented:

```
[match_as_is]
struct CmdMove : Cmd {
    override rtti = "CmdMove"
    x : float
    y : float
}
```

The required is and as operators:

```
def operator is CmdMove ( cmd:Cmd ) {
    return cmd.rtti=="CmdMove"
}

def operator is CmdMove ( anything ) {
    return false
}

def operator as CmdMove ( cmd:Cmd ==const ) : CmdMove const& {
    assert(cmd.rtti=="CmdMove")
    unsafe {
        return reinterpret<CmdMove const&> cmd
    }
}

def operator as CmdMove ( var cmd:Cmd ==const ) : CmdMove& {
    assert(cmd.rtti=="CmdMove")
    unsafe {
```

(continues on next page)

(continued from previous page)

```
        return reinterpret<CmdMove&> cmd
    }
}

def operator as CmdMove ( anything ) {
    panic("Cannot cast to CmdMove")
    return default<CmdMove>
}
```

With these operators in place, you can match against `CmdMove` in a `match` expression:

```
def matching_as_and_is (cmd:Cmd) {
    match ( cmd ) {
        if ( CmdMove(x=$v(x), y=$v(y)) ) {
            return x + y
        }
        if ( _ ) {
            return 0.
        }
    }
}
```

2.30.12 [match_copy] Structure Annotation

The `[match_copy]` annotation provides an alternative to `[match_as_is]` by using a `match_copy` function instead of `is`/`as` operators:

```
[match_copy]
struct CmdLocate : Cmd {
    override rtti = "CmdLocate"
    x : float
    y : float
    z : float
}
```

The `match_copy` function attempts to copy the source into the target type, returning `true` on success:

```
def match_copy ( var cmdm:CmdLocate; cmd:Cmd ) {
    if ( cmd.rtti != "CmdLocate" ) {
        return false
    }
    unsafe {
        cmdm = reinterpret<CmdLocate const&> cmd
    }
    return true
}
```

Usage is identical to regular struct matching:

```
def matching_copy ( cmd:Cmd ) {
    match ( cmd ) {
```

(continues on next page)

(continued from previous page)

```
        if ( CmdLocate(x=$v(x), y=$v(y), z=$v(z)) ) {
            return x + y + z
        }
        if ( _ ) {
            return 0.
        }
    }
}
```

2.30.13 Static Matching

`static_match` works like `match`, but ignores patterns with type mismatches at compile time instead of reporting errors. This makes it suitable for generic functions:

```
static_match ( match_expression ) {
    if ( pattern_1 ) {
        return result_1
    }
    if ( pattern_2 ) {
        return result_2
    }
    ...
    if ( _ ) {
        return result_default
    }
}
```

Example:

```
enum Color {
    red
    green
    blue
}

def enum_static_match ( color, blah ) {
    static_match ( color ) {
        if ( Color red ) {
            return 0
        }
        if ( match_expr(blah) ) {
            return 1
        }
        if ( _ ) {
            return -1
        }
    }
}
```

If `color` is not `Color`, the first case is silently skipped. If `blah` is not `Color`, the second case is skipped. The function always compiles regardless of the argument types.

2.30.14 match_type

The match_type subexpression matches based on the type of an expression:

```
if ( match_type(type<Type>, expr) ) {
    // code to run if match is successful
}
```

Example:

```
def static_match_by_type (what) {
    static_match ( what ) {
        if ( match_type(type<int>,$v(expr)) ) {
            return expr
        }
        if ( _ ) {
            return -1
        }
    }
}
```

If what is of type int, it is bound to expr and returned. Otherwise the catch-all returns -1.

2.30.15 Multi-Match

multi_match evaluates all matching cases instead of stopping at the first match:

```
def multi_match_test ( a:int ) {
    var text = "{a}"
    multi_match ( a ) {
        if ( 0 ) {
            text += " zero"
        }
        if ( 1 ) {
            text += " one"
        }
        if ( 2 ) {
            text += " two"
        }
        if ( $v(a) && (a % 2 == 0) && (a!=0) ) {
            text += " even"
        }
        if ( $v(a) && (a % 2 == 1) ) {
            text += " odd"
        }
    }
    return text
}
```

Unlike match, which stops at the first successful pattern, multi_match continues through all cases. The equivalent code using regular match would require a separate match block for each case.

static_multi_match is a variant of multi_match that works with static_match.

See also:

Variants for the variant type used in `as` matching, *Enumerations* for enumeration types used in enum matching, *Tuples* for tuple types matched by index, *Structs* for struct types matched by field, *Arrays* for dynamic array matching, *Classes* for the `[match_as_is]` annotation, *Expressions* for `is`, `as`, and `?as` operators.

2.31 Annotations

Annotations are metadata decorators attached to functions, structures, classes, enumerations, and variables. They control compiler behavior — export, initialization, safety, optimization, and macro registration.

An annotation is written in square brackets before the declaration it applies to:

```
[export]
def main {
    print("hello\n")
}
```

Multiple annotations can be combined with commas:

```
[export, no_aot]
def main {
    print("hello\n")
}
```

Some annotations accept arguments:

```
[init(tag="db")]
def init_database {
    pass
}

[unused_argument(x, y)]
def foo(x, y : int) {
    pass
}
```

2.31.1 Function Annotations

Lifecycle

[export]
Marks a function as callable from the host application. The host invokes exported functions by name through the context API:

```
[export]
def main {
    print("hello\n")
}
```

[init]
Marks a function to run automatically during context initialization. The function must take no arguments and return `void`:

```
[init]
def setup {
    pass
}
```

Ordering can be controlled with attributes:

- `tag` — defines a named initialization pass
- `before` — runs before the named pass
- `after` — runs after the named pass

All ordering attributes imply late initialization:

```
[init(tag="db")]
def init_database {
    pass
}

[init(after="db")]
def init_cache {
    pass
}
```

[finalize]

Marks a function to run automatically during context shutdown. Same constraints as `[init]` — no arguments, no return value:

```
[finalize]
def cleanup {
    pass
}
```

Supports a `late` attribute for ordering.

[run]

Marks a function to run at compile time:

```
[run]
def compile_time_check {
    print("compiling...\n")
}
```

Disabled by the `disable_run` option (see *Options*).

(see *Program Structure* for the full initialization lifecycle).

Visibility

[private]
Makes a function private to the current module. Equivalent to the `private` keyword before `def`.

Safety

[unsafe_deref]
Marks a function as allowing unsafe dereferences inside its body:

```
[unsafe_deref]
def read_ptr(p : int?) {
    return *p
}
```

[unsafe_operation]
Marks a function as an unsafe operation. Calling it requires an `unsafe` block:

```
[unsafe_operation]
def dangerous_thing {
    pass
}

unsafe {
    dangerous_thing()
}
```

[unsafe_outside_of_for]
Marks a function as unsafe when called outside of a `for` loop body.

Lint Control

[unused_argument]
Suppresses "unused argument" warnings for specific arguments:

```
[unused_argument(x)]
def handler(x : int) {
    pass
}
```

Multiple arguments can be listed: `[unused_argument(x, y)]`.

[nodiscard]
Warns if the return value of the function is discarded:

```
[nodiscard]
def compute : int {
    return 42
}

compute()       // warning: return value discarded
```

[deprecated]
Marks a function as deprecated. Produces a compile-time warning when called:

```
[deprecated(message="use new_func instead")]
def old_func {
    pass
}
```

[no_lint]
Disables all lint checks for this function.

[skip_lock_check]
Skips array/table lock safety checks inside this function.

[sideeffects]
Declares that the function has side effects, even if the compiler cannot detect any. Prevents the optimizer from removing calls to this function.

Generics and Contracts

[generic]
Marks a function as generic (a template that is instantiated for each unique set of argument types):

```
[generic]
def add(a, b : auto) {
    return a + b
}
```

(see *Generic Programming*).

[expect_ref]
Specialization contract: requires named arguments to be references:

```
[expect_ref(arr)]
def process(var arr : auto) {
    pass
}
```

[expect_dim]
Specialization contract: requires named arguments to be fixed-size arrays.

[expect_any_vector]
Specialization contract: requires named arguments to be vector template types.

[local_only]
Verifies that specific arguments are passed as local constructors. The argument value indicates the expected state — `true` means the argument must be a literal constructor, `false` means it must not be:

```
[local_only(data=true)]
def process(data : Foo) {
    pass
}
```

Additional contract annotations are available in `daslib/contracts` (see *Contract Annotations (daslib)* below).

Optimization and AOT

[no_aot]
Disables AOT (ahead-of-time) compilation for this function.

[no_jit]
Disables JIT compilation for this function.

[jit]
Requests JIT compilation for this function.

[hybrid]
Marks a function as an AOT hybrid — it can call interpreted code from AOT context.

[alias_cmres]
Allows aliasing of the copy-on-move return value.

[never_alias_cmres]
Prevents aliasing of the copy-on-move return value.

[pinvoke]
Marks a function for platform invoke (external function call).

[type_function]
Registers the function as usable in type expressions.

Macros

[_macro]
Marks a function as macro initialization code (runs during macro compilation pass).

[macro_function]
Marks a function as existing only in the macro context — excluded from the regular program.

[macro]
Defined in `daslib/ast_boost`. Like `[_macro]` but wraps the function body in a module-ready check. Requires `require daslib/ast_boost`:

```
require daslib/ast_boost

[macro]
def setup_macros {
    print("registering macros\n")
}
```

[tag_function]
Defined in `daslib/ast_boost`. Tags a function with string tags for retrieval via `for_each_tag_function`:

```
[tag_function(my_tag)]
def tagged_func {
    pass
}
```

Markers and Hints

[hint]
A dummy annotation that carries key-value arguments for optimization hints. Does not change behavior by itself.

[marker]
A generic function marker annotation. Does not change behavior.

2.31.2 Structure and Class Annotations

[cpp_layout]
Uses C++ memory layout (matching C++ struct alignment rules):

```
[cpp_layout]
struct CppInterop {
    x : int
    y : float
}
```

Pass `pod=false` to allow non-POD layouts: `[cpp_layout(pod=false)]`.

[safe_when_uninitialized]
Marks the struct as safe even when fields are uninitialized (zero-filled memory is a valid state):

```
[safe_when_uninitialized]
struct Vec2 {
    x : float
    y : float
}
```

[persistent]
Makes a structure persistent (survives context reset). All fields must be POD unless `non_pod=true` is specified:

```
[persistent]
struct Config {
    value : int
}
```

[no_default_initializer]
Suppresses generation of the default constructor for this structure.

[macro_interface]
Marks a structure as a macro interface (for the macro system).

[skip_field_lock_check]
Skips field-level lock checking on array/table fields of this structure.

[comment]
A dummy annotation for attaching comment metadata to a structure.

[tag_structure]
Defined in `daslib/ast_boost`. Tags a structure with string tags for later retrieval.

2.31.3 Macro Registration Annotations (daslib)

These annotations are defined in `daslib/ast_boost` and applied to `class` declarations that inherit from the appropriate AST base class. They auto-register the class as a macro during module compilation.

All accept an optional `name` argument. If omitted, the class name is used.

Annotation	Base class	Purpose
`[function_macro]`	`AstFunctionAnnotation`	Custom function decorator
`[block_macro]`	`AstBlockAnnotation`	Custom block decorator
`[structure_macro]`	`AstStructureAnnotation`	Custom struct/class decorator
`[enumeration_macro]`	`AstEnumerationAnnotation`	Custom enum decorator
`[contract]`	`AstFunctionAnnotation`	Specialization constraint
`[reader_macro]`	`AstReaderMacro`	Custom literal/expression reader
`[comment_reader]`	`AstCommentReader`	Custom comment reader
`[call_macro]`	`AstCallMacro`	Intercepts function-like calls
`[typeinfo_macro]`	`AstTypeInfoMacro`	Custom `typeinfo(...)` handler
`[variant_macro]`	`AstVariantMacro`	Custom variant type processing
`[for_loop_macro]`	`AstForLoopMacro`	Custom for-loop behavior
`[capture_macro]`	`AstCaptureMacro`	Custom closure capture handling
`[type_macro]`	`AstTypeMacro`	Custom type expression processing
`[simulate_macro]`	`AstSimulateMacro`	Custom simulation node generation
`[infer_macro]`	`AstInferMacro`	Runs during type inference
`[dirty_infer_macro]`	`AstDirtyInferMacro`	Runs during dirty inference passes
`[optimization_macro]`	`AstOptimizationMacro`	Runs during optimization
`[lint_macro]`	`AstLintMacro`	Runs during linting
`[global_lint_macro]`	`AstGlobalLintMacro`	Runs after all modules are compiled

Example:

```
require daslib/ast_boost

[function_macro(name="my_decorator")]
class MyDecorator : AstFunctionAnnotation {
    def override apply(var func : FunctionPtr; var group : ModuleGroup;
                       args : AnnotationArgumentList; var errors : das_string) : bool {
        print("decorating {func.name}\n")
        return true
    }
}
```

(see *Macros* for details on writing macros).

2.31.4 Contract Annotations (daslib)

These annotations are defined in `daslib/contracts` and used as specialization constraints on generic function arguments. Each accepts one or more argument names to constrain.

Requires `require daslib/contracts`.

Annotation	Constraint
`[expect_any_array(arg)]`	Argument must be a dynamic array
`[expect_any_enum(arg)]`	Argument must be an enum
`[expect_any_bitfield(arg)]`	Argument must be a bitfield
`[expect_any_vector_type(arg)]`	Argument must be a vector template type
`[expect_any_struct(arg)]`	Argument must be a struct
`[expect_any_numeric(arg)]`	Argument must be a numeric type
`[expect_any_workhorse(arg)]`	Argument must be a “workhorse” type (int, float, etc.)
`[expect_any_tuple(arg)]`	Argument must be a tuple
`[expect_any_variant(arg)]`	Argument must be a variant
`[expect_any_function(arg)]`	Argument must be a function type
`[expect_any_lambda(arg)]`	Argument must be a lambda
`[expect_ref(arg)]`	Argument must be a reference
`[expect_pointer(arg)]`	Argument must be a pointer
`[expect_class(arg)]`	Argument must be a class pointer
`[expect_value_handle(arg)]`	Argument must be a value handle type

Example:

```
require daslib/contracts

[expect_any_array(arr)]
def first_element(arr : auto) {
    return arr[0]
}
```

2.31.5 Annotation Syntax Details

Annotations can be combined with logical operators for contract composition:

```
[expect_ref(a) && expect_dim(b)]
def process(var a : auto; b : auto) {
    pass
}
```

Negation is also supported:

```
[!expect_ref(a)]
def no_ref(a : auto) {
    pass
}
```

Annotations on struct/class fields appear before the field name in the `@` metadata syntax:

```
class Foo {
    @big
    @min = 13
    @max = 42
    value : int
}
```

These `@` decorators attach metadata to the field. They are accessible via `typeinfo` and at compile time in macros.

2.32 Options

The `options` statement sets compilation options for the current file. Options control compiler behavior — linting, optimization, logging, memory limits, and language features.

Syntax:

```
options name = value
options name                    // shorthand for name = true
options a = true, b = false     // multiple options, comma-separated
```

Option values can be `bool`, `int`, or `string`, depending on the option.

Multiple `options` statements are allowed per file. They can appear before or after `module` and `require` declarations. By convention they are placed at the top of the file:

```
options gen2
options no_unused_block_arguments = false

module my_module shared public

require daslib/strings_boost
```

Note: The host application can restrict which options are allowed in which files via the `isOptionAllowed` method on the project's file access object.

2.32.1 Language and Syntax

Option	Type	Default	Description
`gen2`	bool	false	Enables generation-2 syntax. Requires `{ }` braces for all blocks, `;` semicolons, and disables legacy make syntax (`[[...]]` for structs, `[{...}]` for arrays). Applied immediately during parsing.
`indenting`	int	0	Tab size for indentation-sensitive parsing. Valid values are `0` (disabled), `2`, `4`, `8`. Applied immediately during parsing.
`always_export_initializer`	bool	false	Always exports initializer functions.
`infer_time_folding`	bool	true	Enables constant folding during type inference.
`disable_run`	bool	false	Disables compile-time execution (the `[run]` annotation has no effect).

2.32.2 Lint Control

Option	Type	Default	Description
`lint`	bool	true	Enables or disables all lint checks for this module.
`no_writing_to_nameless`	bool	true	Disallows writing to nameless (intermediate) variables on the stack.
`always_call_super`	bool	false	Requires `super()` to be called in every class constructor.
`no_unused_function_arguments`	bool	false	Reports unused function arguments as errors.
`no_unused_block_arguments`	bool	false	Reports unused block arguments as errors.
`no_deprecated`	bool	false	Reports use of deprecated features as errors.
`no_aliasing`	bool	false	Reports aliasing as errors. When `false`, aliasing only disables certain optimizations.
`strict_smart_pointers`	bool	true	Enables stricter safety checks for smart pointers.
`strict_properties`	bool	false	Enables strict property semantics. When `true`, `a.prop = b` is not promoted to `a.prop := b`.
`report_invisible_functions`	bool	true	Reports functions that are not visible from the current module.
`report_private_functions`	bool	true	Reports functions that are inaccessible due to private module visibility.

2.32.3 Optimization

Option	Type	Default	Description
`optimize`	bool	true	Enables compiler optimizations.
`no_optimization`	bool	false	Disables compiler optimizations (inverse of `optimize`).
`no_optimizations`	bool	false	Disables all optimizations unconditionally. Overrides all other optimization settings.
`fusion`	bool	true	Enables simulation node fusion for faster execution.
`remove_unused_symbols`	bool	true	Removes unused symbols from the compiled program. Must be `false` for modules compiled for AOT, otherwise the AOT tool will not find all necessary symbols.
`no_fast_call`	bool	false	Disables the fastcall optimization.

2.32.4 Memory

Option	Type	Default	Description
stack	int	16384	Stack size in bytes. Set to 0 for a unique stack per context.
heap_size_hint	int	65536	Initial heap size hint in bytes.
string_heap_size_hint	int	65536	Initial string heap size hint in bytes.
heap_size_limit	int	0	Maximum heap allocation in bytes. 0 means unlimited.
string_heap_size_limit	int	0	Maximum string heap allocation in bytes. 0 means unlimited.
persistent_heap	bool	false	Uses a persistent (non-releasing) heap. Old allocations are never freed — useful with garbage collection.
gc	bool	false	Enables garbage collection for the context.
intern_strings	bool	false	Enables string interning lookup for the regular string heap.
very_safe_context	bool	false	Context does not release old memory from array or table growth (leaves it to the GC).

2.32.5 AOT

Option	Type	Default	Description
no_aot	bool	false	Disables AOT (ahead-of-time) compilation for this module.
aot_prologue	bool	false	Generates AOT prologue code (enables AOT export even when not strictly required).
aot_lib	bool	false	Compiles module in AOT library mode.
standalone_context	bool	false	Generates a standalone context class in AOT mode.
only_fast_aot	bool	false	Only allows fast AOT functions.
aot_order_side_effects	bool	false	Orders AOT side effects for deterministic evaluation.

2.32.6 Safety and Strictness

Option	Type	Default	Description
no_init	bool	false	Disallows [init] functions in any form.
no_global_variables	bool	false	Disallows module-level global variables (shared variables are still allowed).
no_global_variables_at_all	bool	false	Disallows all global variables, including shared ones.
no_global_heap	bool	false	Disallows global heap allocation.
no_unsafe_uninitialized_structures	bool	true	Disallows unsafe-ly uninitialized structures.
unsafe_table_lookup	bool	true	Makes table lookup (tab[key]) an unsafe operation.
relaxed_pointer_const	bool	false	Relaxes const correctness on pointers.
relaxed_assign	bool	true	Allows the = operator to be silently promoted to <- when the right-hand side is a temporary value or function call result and the type cannot be copied. See *Move, Copy, and Clone*.
strict_unsafe_delete	bool	false	Makes delete of types that contain unsafe delete an unsafe operation.
no_local_class_members	bool	true	Struct and class members cannot be class types.
skip_lock_checks	bool	false	Skips all lock safety checks at runtime.
skip_module_lock_checks	bool	false	Skips lock checks for this module specifically.

2.32.7 Multiple Contexts

Option	Type	Default	Description
multiple_contexts	bool	false	Indicates that the code supports multiple context safety.
solid_context	bool	false	All variable and function lookup is context-dependent (via index). Slightly faster, but prohibits AOT and patches.

2.32.8 Debugging and Profiling

Option	Type	Default	Description
debugger	bool	false	Enables debugger support. Disables fastcall and adds a context mutex.
profiler	bool	false	Enables profiler support. Disables fastcall.
debug_infer_flag	bool	false	Enables debugging of macro “not_inferred” issues.
threadlock_context	bool	false	Adds a mutex to the context for thread-safe access.
log_compile_time	bool	false	Prints compile time at the end of compilation.
log_total_compile_time	bool	false	Prints detailed compile time breakdown at the end of compilation.

2.32.9 RTTI

Option	Type	Default	Description
`rtti`	bool	false	Creates extended RTTI (runtime type information). Required for reflection, `typeinfo`, and the `rtti` module.

2.32.10 Logging

The following options control diagnostic output during compilation. All are `bool` type, default `false`.

Option	Description
`log`	Logs the program AST after compilation.
`log_optimization_passes`	Logs each optimization pass.
`log_optimization`	Logs optimization results.
`log_stack`	Logs stack allocation.
`log_init`	Logs initialization.
`log_symbol_use`	Logs symbol usage information.
`log_var_scope`	Logs variable scoping.
`log_nodes`	Logs simulation nodes.
`log_nodes_aot_hash`	Logs AOT hash values for simulation nodes.
`log_mem`	Logs memory usage after simulation.
`log_debug_mem`	Logs debug memory information.
`log_cpp`	Logs C++ representation.
`log_aot`	Logs AOT output.
`log_infer_passes`	Logs type inference passes.
`log_require`	Logs module resolution.
`log_generics`	Logs generic function instantiation.
`log_mn_hash`	Logs mangled name hashes.
`log_gmn_hash`	Logs global mangled name hashes.
`log_ad_hash`	Logs annotation data hashes.
`log_aliasing`	Logs aliasing analysis.
`log_inscope_pod`	Logs inscope analysis for POD-like types.

2.32.11 Print Control

The following options affect how the AST is printed (for `log` output). All are `bool` type, default `false`.

Option	Description
`print_ref`	Prints reference information in AST output.
`print_var_access`	Prints variable access flags in AST output.
`print_c_style`	Uses C-style formatting for AST output.
`print_use`	Prints symbol usage flags.

2.32.12 Miscellaneous

Option	Type	Default	Description
`max_infer_passes`	int	50	Maximum number of type inference passes before the compiler gives up.
`scoped_stack_allocator`	bool	true	Reuses stack memory after variables go out of scope.
`force_inscope_pod`	bool	false	Forces inscope lifetime semantics for POD-like types.
`keep_alive`	bool	false	Produces keep-alive nodes in simulation.
`serialize_main_module`	bool	true	Serializes the main module instead of recompiling it each time.

2.32.13 Module-Registered Options

Any module can register custom options via its `getOptionType` method. These are validated during lint alongside the built-in options. If a module defines its own option, any file that requires that module can use it.

If you set an option that is not recognized by either the built-in list, the policy list, or any loaded module, the compiler reports:

```
invalid option 'unknown_name'
```

If the option exists but you provide the wrong value type, the compiler reports:

```
invalid option type for 'name', unexpected 'float', expecting 'bool'
```

See also:

Contexts for memory and stack allocation options, *Annotations* for annotation-based equivalents, *Locks* for `skip_lock_checks` option, *Unsafe* for `unsafe_table_lookup` option, *Program structure* for overall file layout and `options` placement.

2.33 Macros

In Daslang, macros are the machinery that allow direct manipulation of the syntax tree.

Macros are exposed via the daslib/ast module and daslib/ast_boost helper module.

Macros are evaluated at compilation time during different compilation passes. Macros assigned to a specific module are evaluated as part of the module every time that module is included.

2.33.1 Compilation passes

The Daslang compiler performs compilation passes in the following order for each module (see *Modules*):

1. Parser transforms das program to AST
    1. If there are any parsing errors, compilation stops
2. `apply` is called for every function or structure
    1. If there are any errors, compilation stops
3. Infer pass repeats itself until no new transformations are reported
    1. Built-in infer pass happens

        1. `transform` macros are called for every function or expression
    2. Macro passes happen
4. If there are still any errors left, compilation stops
5. `finish` is called for all functions and structure macros
6. Lint pass happens
    1. If there are any errors, compilation stops
7. Optimization pass repeats itself until no new transformations are reported
    1. Built-in optimization pass happens
    2. Macro optimization pass happens
8. If there are any errors during optimization passes, compilation stops
9. If the module contains any macros, simulation happens
    1. If there are any simulation errors, compilation stops
    2. Module macro functions (annotated with `_macro`) are invoked
        1. If there are any errors, compilation stops

Modules are compiled in `require` order.

2.33.2 Invoking macros

The `[_macro]` annotation is used to specify functions that should be evaluated at compilation time . Consider the following example from daslib/ast_boost:

```
[_macro,private]
def setup {
    if ( is_compiling_macros_in_module("ast_boost") ) {
        add_new_function_annotation("macro", new MacroMacro())
    }
}
```

The `setup` function is evaluated after the compilation of each module, which includes ast_boost. The `is_compiling_macros_in_module` function returns true if the currently compiled module name matches the argument. In this particular example, the function annotation `macro` would only be added once: when the module `ast_boost` is compiled.

Macros are invoked in the following fashion:

1. Class is derived from the appropriate base macro class
2. Adapter is created
3. Adapter is registered with the module

For example, this is how this lifetime cycle is implemented for the reader macro:

```
def add_new_reader_macro ( name:string; someClassPtr ) {
    var ann <- make_reader_macro(name, someClassPtr)
    this_module() |> add_reader_macro(ann)
    unsafe {
        delete ann
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.33.3 AstFunctionAnnotation

The `AstFunctionAnnotation` macro allows you to manipulate calls to specific functions as well as their function bodies. Annotations can be added to regular or generic functions.

`add_new_function_annotation` adds a function annotation to a module. There is additionally the `[function_macro]` annotation which accomplishes the same thing.

`AstFunctionAnnotation` allows several different manipulations:

```
class AstFunctionAnnotation {
    def abstract transform ( var call : smart_ptr<ExprCallFunc>; var errors : das_string␣
↪) : ExpressionPtr
    def abstract verifyCall ( var call : smart_ptr<ExprCallFunc>; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract apply ( var func:FunctionPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract patch ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string; var astChanged:bool& ) : bool
    def abstract fixup ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract lint ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract complete ( var func:FunctionPtr; var ctx:smart_ptr<Context> ) : void
    def abstract isCompatible ( var func:FunctionPtr; var types:VectorTypeDeclPtr;␣
↪decl:AnnotationDeclaration; var errors:das_string ) : bool
    def abstract isSpecialized : bool
    def abstract appendToMangledName ( func:FunctionPtr; decl:AnnotationDeclaration; var␣
↪mangledName:das_string ) : void
}
```

`transform` lets you change calls to the function and is applied at the infer pass. Transform is the best way to replace or modify function calls with other semantics.

`verifyCall` is called during the `lint` phase on each call to the function and is used to check if the call is valid.

`apply` is applied to the function itself before the infer pass. Apply is typically where global function body modifications or instancing occurs.

`finish` is applied to the function itself after the infer pass. It's only called on non-generic functions or instances of the generic functions. `finish` is typically used to register functions, notify C++ code, etc. After this, the function is fully defined and inferred, and can no longer be modified.

`patch` is called after the infer pass. If patch sets astChanged to true, the infer pass will be repeated.

`fixup` is called after the infer pass. It's used to fixup the function's body.

`lint` is called during the `lint` phase on the function itself and is used to verify that the function is valid.

`complete` is called during the `simulate` portion of context creation. At this point Context is available.

`isSpecialized` must return true if the particular function matching is governed by contracts. In that case, `isCompatible` is called, and the result taken into account.

`isCompatible` returns true if a specialized function is compatible with the given arguments. If a function is not compatible, the errors field must be specified.

`appendToMangledName` is called to append a mangled name to the function. That way multiple functions with the same type signature can exist and be differentiated between.

Lets review the following example from `ast_boost` of how the `macro` annotation is implemented:

```
class MacroMacro : AstFunctionAnnotation {
    def override apply ( var func:FunctionPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool {
        compiling_program().flags |= ProgramFlags needMacroModule
        func.flags |= FunctionFlags init
        var blk <- new ExprBlock(at=func.at)
        var ifm <- new ExprCall(at=func.at, name:="is_compiling_macros")
        var ife <- new ExprIfThenElse(at=func.at, cond<-ifm, if_true<-func.body)
        push(blk.list,ife)
        func.body <- blk
        return true
    }
}
```

During the `apply` pass the function body is appended with the `if is_compiling_macros()` closure. Additionally, the `init` flag is set, which is equivalent to a `_macro` annotation. Functions annotated with `[macro]` are evaluated during module compilation.

2.33.4 AstBlockAnnotation

`AstBlockAnnotation` is used to manipulate block expressions (blocks, lambdas, local functions):

```
class AstBlockAnnotation {
    def abstract apply ( var blk:smart_ptr<ExprBlock>; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var blk:smart_ptr<ExprBlock>; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
}
```

`add_new_block_annotation` adds a block annotation to a module. There is additionally the `[block_macro]` annotation which accomplishes the same thing.

`apply` is called for every block expression before the infer pass.

`finish` is called for every block expression after infer pass.

2.33.5 AstStructureAnnotation

The `AstStructureAnnotation` macro lets you manipulate structure or class definitions via annotation:

```
class AstStructureAnnotation {
    def abstract apply ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract patch ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string; var astChanged:bool& ) : bool
    def abstract complete ( var st:StructurePtr; var ctx:smart_ptr<Context> ) : void
}
```

`add_new_structure_annotation` adds a structure annotation to a module. There is additionally the `[structure_macro]` annotation which accomplishes the same thing.

`AstStructureAnnotation` allows 4 different manipulations:

```
class AstStructureAnnotation {
    def abstract apply ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract patch ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string; var astChanged : bool& ) : bool
    def abstract complete ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string; var context : smart_ptr<Context>␣
↪) : bool
}
```

`apply` is invoked before the infer pass. It is the best time to modify the structure, generate some code, etc.

`finish` is invoked after the successful infer pass. Its typically used to register structures, perform RTTI operations, etc. After this, the structure is fully inferred and defined and can no longer be modified afterwards.

`patch` is invoked after the infer pass. If patch sets astChanged to true, the infer pass will be repeated.

`complete` is invoked during the `simulate` portion of context creation. At this point Context is available.

An example of such annotation is `SetupAnyAnnotation` from daslib/ast_boost.

2.33.6 AstEnumerationAnnotation

The `AstEnumerationAnnotation` macro lets you manipulate enumerations via annotation:

```
class AstEnumerationAnnotation {
    def abstract apply ( var st:EnumerationPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
}
```

`add_new_enumeration_annotation` adds an enumeration annotation to a module. There is additionally the `[enumeration_macro]` annotation which accomplishes the same thing.

`apply` is invoked before the infer pass. It is the best time to modify the enumeration, generate some code, etc.

In gen2 syntax, register an enumeration macro and annotate enums with:

```
[enumeration_macro(name="enum_total")]
class EnumTotalAnnotation : AstEnumerationAnnotation {
    def override apply(var enu : EnumerationPtr; var group : ModuleGroup;
                       args : AnnotationArgumentList;
                       var errors : das_string) : bool {
        // modify enu.list or generate code
        return true
    }
}

[enum_total]
enum Direction { North; South; East; West }
```

See also:

Tutorial: tutorial_macro_enumeration_macro — step-by-step enumeration macro examples (enum modification and code generation)

Standard library: `daslib/enum_trait.das` — enum_trait module reference

2.33.7 AstVariantMacro

`AstVariantMacro` is specialized in transforming `is`, `as`, and `?as` expressions.

`add_new_variant_macro` adds a variant macro to a module. There is additionally the `[variant_macro]` annotation which accomplishes the same thing.

Each of the 3 transformations are covered in the appropriate abstract function:

```
class AstVariantMacro {
    def abstract visitExprIsVariant     ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprIsVariant> ) : ExpressionPtr
    def abstract visitExprAsVariant     ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprAsVariant> ) : ExpressionPtr
    def abstract visitExprSafeAsVariant ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprSafeAsVariant> ) : ExpressionPtr
}
```

Let's review the following example from daslib/ast_boost:

```
// replacing ExprIsVariant(value,name) => ExprOp2("==",value.__rtti,"name")
// if value is ast::Expr*
class BetterRttiVisitor : AstVariantMacro {
    def override visitExprIsVariant(prog:ProgramPtr; mod:Module?;expr:smart_ptr
↪<ExprIsVariant>) : ExpressionPtr {
        if ( isExpression(expr.value._type) ) {
            var vdr <- new ExprField(at=expr.at, name:="__rtti", value <- clone_
↪expression(expr.value))
            var cna <- new ExprConstString(at=expr.at, value:=expr.name)
            var veq <- new ExprOp2(at=expr.at, op:="==", left<-vdr, right<-cna)
            return veq
        }
        return default<ExpressionPtr>
    }
```

(continues on next page)

(continued from previous page)

```
}

// note the following usage
class GetHintFnMacro : AstFunctionAnnotation {
    def override transform ( var call : smart_ptr<ExprCall>; var errors : das_string ) :␣
↪ExpressionPtr {
        if ( call.arguments[1] is ExprConstString ) {    // HERE EXPRESSION WILL BE␣
↪REPLACED
            ...
        }
    }
}
```

Here, the macro takes advantage of the ExprIsVariant syntax. It replaces the `expr is TYPENAME` expression with an `expr.__rtti = "TYPENAME"` expression. The `isExpression` function ensures that `expr` is from the `ast::Expr*` family, i.e. part of the Daslang syntax tree.

2.33.8 AstReaderMacro

`AstReaderMacro` allows embedding a completely different syntax inside Daslang code.

`add_new_reader_macro` adds a reader macro to a module. There is additionally the `[reader_macro]` annotation, which essentially automates the same thing.

Reader macros accept characters, collect them if necessary, and produce output via one of two patterns:

```
class AstReaderMacro {
    def abstract accept ( prog:ProgramPtr; mod:Module?; expr:ExprReader?; ch:int;␣
↪info:LineInfo ) : bool
    def abstract visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprReader> ) :␣
↪ExpressionPtr
    def abstract suffix ( prog:ProgramPtr; mod:Module?; expr:ExprReader?; info:LineInfo;␣
↪var outLine:int&; var outFile:FileInfo?& ) : string
}
```

Reader macros are invoked via the `% READER_MACRO_NAME ~ character_sequence` syntax. The `accept` function notifies the correct terminator of the character sequence:

```
var x = %arr~\{\}\w\x\y\n%% // invoking reader macro arr, %% is a terminator
```

Consider the implementation for the example above:

```
[reader_macro(name="arr")]
class ArrayReader : AstReaderMacro {
    def override accept ( prog:ProgramPtr; mod:Module?; var expr:ExprReader?; ch:int;␣
↪info:LineInfo ) : bool {
        append(expr.sequence,ch)
        if ( ends_with(expr.sequence,"%%") ) {
            let len = length(expr.sequence)
            resize(expr.sequence,len-2)
            return false
        } else {
            return true
```

(continues on next page)

(continued from previous page)

```
        }
    }
    def override visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprReader> ) :␣
↪ExpressionPtr {
        let seqStr = string(expr.sequence)
        var arrT <- new TypeDecl(baseType=Type tInt)
        push(arrT.dim,length(seqStr))
        var mkArr <- new ExprMakeArray(at = expr.at, makeType <- arrT)
        for ( x in seqStr ) {
            var mkC <- new ExprConstInt(at=expr.at, value=x)
            push(mkArr.values,mkC)
        }
        return <- mkArr
    }
}
```

The `accept` function macro collects symbols in the sequence. Once the sequence ends with the terminator sequence %%, `accept` returns false to indicate the end of the sequence.

In `visit`, the collected sequence is converted into a make array `[ch1,ch2,..]` expression.

More complex examples include the JsonReader macro in daslib/json_boost or RegexReader in daslib/regex_boost.

`suffix` is an alternative to `visit` — it is called immediately after `accept` during parsing, before the AST is built. Instead of returning an AST node, it returns a **string** of daScript source code that the parser re-parses. This is useful for generating top-level declarations (functions, structs) from custom syntax. When used at module level the `ExprReader` node is discarded, and the suffix text is the only output. `SpoofInstanceReader` in `daslib/spoof.das` is an example.

The `outLine` and `outFile` parameters allow remapping line information for error reporting in the injected code.

See also:

Tutorial: Reader Macros — step-by-step example of both visit and suffix patterns.

2.33.9 AstCallMacro

`AstCallMacro` operates on expressions which have function call syntax or something similar. It occurs during the infer pass.

`add_new_call_macro` adds a call macro to a module. The `[call_macro]` annotation automates the same thing:

```
class AstCallMacro {
    def abstract preVisit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprCallMacro> ) : void
    def abstract visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprCallMacro> ) : ExpressionPtr
    def abstract canVisitArguments ( expr:smart_ptr<ExprCallMacro> ) : bool
}
```

`apply` from daslib/apply is an example of such a macro:

```
[call_macro(name="apply")]  // apply(value, block)
class ApplyMacro : AstCallMacro {
    def override visit ( prog:ProgramPtr; mod:Module?; var expr:smart_ptr<ExprCallMacro>␣
↪) : ExpressionPtr {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

Note how the name is provided in the `[call_macro]` annotation.

`preVisit` is called before the arguments are visited.

`visit` is called after the arguments are visited.

`canVisitArguments` is called to determine if the macro can visit the arguments.

2.33.10 AstPassMacro

`AstPassMacro` is one macro to rule them all. It gets the entire program as input and can be invoked at numerous passes:

```
class AstPassMacro {
    def abstract apply(prog : ProgramPtr; mod : Module?) : bool
}
```

Five annotations control when a pass macro runs:

- `[infer_macro]` — after clean type inference. Returning `true` re-infers.
- `[dirty_infer_macro]` — during each dirty inference pass.
- `[lint_macro]` — after successful compilation (lint phase, read-only).
- `[global_lint_macro]` — same as `[lint_macro]` but for all modules.
- `[optimization_macro]` — during the optimisation loop.

`make_pass_macro` registers a class as a pass macro.

Typically, such macros create an `AstVisitor` which performs the necessary transformations via `visit(prog, adapter)`.

See also:

tutorial_macro_pass_macro — step-by-step tutorial with lint and infer macro examples.

2.33.11 AstTypeMacro

`AstTypeMacro` lets you define custom type expressions resolved during type inference. It has a single method:

```
class AstTypeMacro {
    def abstract visit ( prog:ProgramPtr; mod:Module?; td:TypeDeclPtr; passT:TypeDeclPtr␣
↪) : TypeDeclPtr
}
```

`add_new_type_macro` adds a type macro to a module. The `[type_macro(name="...")]` annotation automates registration.

The compiler parses invocations like `name(type<T>, N)` in type position into a `TypeDecl` with `baseType = Type.typeMacro`. The arguments are stored in `td.dimExpr`:

- `dimExpr[0]` — `ExprConstString` with the macro name

- `dimExpr[1..]` — user arguments (`ExprTypeDecl` for types, `ExprConstInt` for integers, etc.)

`visit()` is called in two contexts:

- **Concrete** — all types are inferred; `passT` is null; `dimExpr[i]._type` is the resolved type.
- **Generic** — type parameters like `auto(TT)` are unresolved; `passT` carries the actual argument type for matching; `dimExpr[i]._type` is null.

See also:

tutorial_macro_type_macro — step-by-step tutorial showing concrete and generic type-macro usage.

2.33.12 AstTypeInfoMacro

`AstTypeInfoMacro` is designed to implement custom type information inside a typeinfo expression:

```
class AstTypeInfoMacro {
    def abstract getAstChange ( expr:smart_ptr<ExprTypeInfo>; var errors:das_string ) :␣
↪ExpressionPtr
    def abstract getAstType ( var lib:ModuleLibrary; expr:smart_ptr<ExprTypeInfo>; var␣
↪errors:das_string ) : TypeDeclPtr
}
```

`add_new_typeinfo_macro` adds a typeinfo macro to a module. There is additionally the `[typeinfo_macro]` annotation, which essentially automates the same thing.

The `typeinfo` expression uses gen2 syntax with the trait name **outside** the parentheses:

```
typeinfo trait_name(type<T>)                // basic
typeinfo trait_name<subtrait>(type<T>)      // with subtrait
typeinfo trait_name<sub;extra>(type<T>)     // with subtrait and extratrait
```

`getAstChange` returns a newly generated AST node for the typeinfo expression. Alternatively, it returns null if no changes are required, or if there is an error. In case of error, the errors string must be filled.

`getAstType` returns the type of the new typeinfo expression.

See also:

Tutorial: tutorial_macro_typeinfo_macro — step-by-step guide with three `getAstChange` examples (struct description, enum names, method check).

2.33.13 AstForLoopMacro

`AstForLoopMacro` is designed to implement custom processing of for loop expressions:

```
class AstForLoopMacro {
    def abstract visitExprFor ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprFor> )␣
↪: ExpressionPtr
}
```

`add_new_for_loop_macro` adds a reader macro to a module. There is additionally the `[for_loop_macro]` annotation, which essentially automates the same thing.

`visitExprFor` is similar to that of `AstVisitor`. It returns a new expression, or null if no changes are required.

2.33.14 AstCaptureMacro

AstCaptureMacro is designed to implement custom capturing and finalization of lambda expressions:

```
class AstCaptureMacro {
    def abstract captureExpression ( prog:Program?; mod:Module?; expr:ExpressionPtr;␣
↪etype:TypeDeclPtr ) : ExpressionPtr
    def abstract captureFunction ( prog:Program?; mod:Module?; var lcs:Structure?; var␣
↪fun:FunctionPtr ) : void
    def abstract releaseFunction ( prog:Program?; mod:Module?; var lcs:Structure?; var␣
↪fun:FunctionPtr ) : void
}
```

add_new_capture_macro adds a reader macro to a module. There is additionally the [capture_macro] annotation, which essentially automates the same thing.

captureExpression is called per captured variable when the lambda struct is being built. It returns a replacement expression to wrap the capture, or null if no changes are required.

captureFunction is called once after the lambda function is generated. Use this to inspect captured fields (lcs) and append code to (fun.body as ExprBlock).finalList — which runs **after each invocation** (per-call finally), not on destruction.

releaseFunction is called once when the lambda **finalizer** is generated. fun is the finalizer function (not the lambda call function). Code appended to (fun.body as ExprBlock).list runs on **destruction** — after the user-written finally {} block but before the compiler-generated field cleanup (delete *__this).

See also:

Tutorial: Capture Macros — step-by-step example using all three hooks with an [audited] tag annotation.

2.33.15 AstCommentReader

AstCommentReader is designed to implement custom processing of comment expressions:

```
class AstCommentReader {
    def abstract open ( prog:ProgramPtr; mod:Module?; cpp:bool; info:LineInfo ) : void
    def abstract accept ( prog:ProgramPtr; mod:Module?; ch:int; info:LineInfo ) : void
    def abstract close ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract beforeStructure ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterStructure ( st:StructurePtr; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeStructureFields ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract afterStructureField ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract afterStructureFields ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract beforeFunction ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterFunction ( fn:FunctionPtr; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeGlobalVariables ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract afterGlobalVariable ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
```

(continues on next page)

(continued from previous page)

```
    def abstract afterGlobalVariables ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract beforeVariant ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterVariant ( name:string; prog:ProgramPtr; mod:Module?; info:LineInfo␣
↪) : void
    def abstract beforeEnumeration ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterEnumeration ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeAlias ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterAlias ( name:string; prog:ProgramPtr; mod:Module?; info:LineInfo )␣
↪: void
}
```

`add_new_comment_reader` adds a reader macro to a module. There is additionally the `[comment_reader]` annotation, which essentially automates the same thing.

`open` occurs when the parsing of a comment starts.

`accept` occurs for every character of the comment.

`close` occurs when a comment is over.

`beforeStructure` and `afterStructure` occur before and after each structure or class declaration, regardless of if it has comments.

`beforeStructureFields` and `afterStructureFields` occur before and after each structure or class field, regardless of if it has comments.

`afterStructureField` occurs after each field declaration.

`beforeFunction` and `afterFunction` occur before and after each function declaration, regardless of if it has comments.

`beforeGlobalVariables` and `afterGlobalVariables` occur before and after each global variable declaration, regardless of if it has comments.

`afterGlobalVariable` occurs after each individual global variable declaration.

`beforeVariant` and `afterVariant` occur before and after each variant declaration, regardless of if it has comments.

`beforeEnumeration` and `afterEnumeration` occur before and after each enumeration declaration, regardless of if it has comments.

`beforeAlias` and `afterAlias` occur before and after each alias type declaration, regardless or if it has comments.

2.33.16 AstSimulateMacro

`AstSimulateMacro` is designed to customize the simulation of the program:

```
class AstSimulateMacro {
    def abstract preSimulate ( prog:Program?; ctx:Context? ) : bool
    def abstract simulate ( prog:Program?; ctx:Context? ) : bool
}
```

`preSimulate` occurs after the context has been simulated, but before all the structure and function annotation simulations.

`simulate` occurs after all the structure and function annotation simulations.

2.33.17 AstVisitor

`AstVisitor` implements the visitor pattern for the Daslang expression tree. It contains a callback for every single expression in prefix and postfix form, as well as some additional callbacks:

```
class AstVisitor {
    ...
    // find
        def abstract preVisitExprFind(expr:smart_ptr<ExprFind>) : void          // prefix
        def abstract visitExprFind(expr:smart_ptr<ExprFind>) : ExpressionPtr    //␣
↪postifx
    ...
}
```

Postfix callbacks can return expressions to replace the ones passed to the callback.

PrintVisitor from the `ast_print` example implements the printing of every single expression in Daslang syntax.

`make_visitor` creates a visitor adapter from the class, derived from `AstVisitor`. The adapter then can be applied to a program via the `visit` function:

```
var astVisitor = new PrintVisitor()
var astVisitorAdapter <- make_visitor(*astVisitor)
visit(this_program(), astVisitorAdapter)
```

If an expression needs to be visited, and can potentially be fully substituted, the `visit_expression` function should be used:

```
expr <- visit_expression(expr,astVisitorAdapter)
```

See also:

Annotations for annotation-based macro registration, *Reification* for AST reification used in macros, *Program structure* for the compilation lifecycle, *Generic programming* for `typeinfo` macros.

2.34 Reification

Expression reification is used to generate AST expression trees in a convenient way. It provides a collection of escaping sequences to allow for different types of expression substitutions. At the top level, reification is supported by multiple call macros, which are used to generate different AST objects.

Reification is implemented in daslib/templates_boost.

2.34.1 Simple example

Let's review the following example:

```
var foo = "foo"
var fun <- qmacro_function("madd") <| $ ( a, b ) {
    return $i(foo) * a + b
}
print(describe(fun))
```

The output would be:

```
def public madd (  a:auto const;  b:auto const ) : auto {
    return (foo * a) + b
}
```

What happens here is that call to macro `qmacro_function` generates a new function named `madd`. The arguments and body of that function are taken from the block, which is passed to the function. The escape sequence $i takes its argument in the form of a string and converts it to an identifier (ExprVar).

2.34.2 Quote macros

Reification macros are similar to `quote` expressions because the arguments are not going through type inference. Instead, an ast tree is generated and operated on.

qmacro

`qmacro` is the simplest reification. The input is returned as is, after escape sequences are resolved:

```
var expr <- qmacro(2+2)
print(describe(expr))
```

prints:

```
(2+2)
```

qmacro_block

`qmacro_block` takes a block as an input and returns unquoted block. To illustrate the difference between `qmacro` and `qmacro_block`, let's review the following example:

```
var blk1 <- qmacro <| $ ( a, b ) { return a+b; }
var blk2 <- qmacro_block <| $ ( a, b ) { return a+b; }
print("{blk1.__rtti}\n{blk2.__rtti}\n")
```

The output would be:

```
ExprMakeBlock
ExprBlock
```

This is because the block sub-expression is decorated, i.e. (ExprMakeBlock(ExprBlock (...))), and `qmacro_block` removes such decoration.

qmacro_expr

`qmacro_expr` takes a block with a single expression as an input and returns that expression as the result. Certain expressions like `return` and such can't be an argument to a call, so they can't be passed to `qmacro` directly. The work around is to pass them as first line of a block:

```
var expr <- qmacro_block() {
    return 13
}
print(describe(expr))
```

prints:

```
return 13
```

qmacro_type

qmacro_type takes a type expression (type<...>) as an input and returns the subtype as a TypeDeclPtr, after resolving the escape sequences. Consider the following example:

```
var foo <- typeinfo ast_typedecl(type<int>)
var typ <- qmacro_type <| type<$t(foo)?>
print(describe(typ))
```

TypeDeclPtr foo is passed as a reification sequence to qmacro_type, and a new pointer type is generated. The output is:

```
int?
```

qmacro_function

qmacro_function takes two arguments. The first one is the generated function name. The second one is a block with a function body and arguments. By default, the generated function only has the *FunctionFlags generated* flag set.

qmacro_variable

qmacro_variable takes a variable name and type expression as an input, and returns the variable as a VariableDeclPtr, after resolving the escape sequences:

```
var vdecl <- qmacro_variable("foo", type<int>)
print(describe(vdecl))
```

prints:

```
foo:int
```

2.34.3 Escape sequences

Reification provides multiple escape sequences, which are used for miscellaneous template substitution.

$i(ident)

$i takes a string or das_string as an argument and substitutes it with an identifier. An identifier can be substituted for the variable name in both the variable declaration and use:

```
var bus = "bus"
var qb <- qmacro_block() {
    let $i(bus) = "busbus"
    let t = $i(bus)
}
print(describe(qb))
```

prints:

```
let  bus:auto const = "busbus"
let  t:auto const = bus
```

$f(field-name)

$f takes a string or das_string as an argument and substitutes it with a field name:

```
var bar = "fieldname"
var blk <- qmacro_block() {
    foo.$f(bar) = 13
}
print(describe(blk))
```

prints:

```
foo.fieldname = 13
```

$v(value)

$v takes any value as an argument and substitutes it with an expression which generates that value. The value does not have to be a constant expression, but the expression will be evaluated before its substituted. Appropriate make infrastructure will be generated:

```
var t = (1,2.,"3")
var expr <- qmacro($v(t))
print(describe(expr))
```

prints:

```
(1,2f,"3")
```

In the example above, a tuple is substituted with the expression that generates this tuple.

$e(expression)

$e takes any expression as an argument in form of an ExpressionPtr. The expression will be substituted as-is:

```
var expr <- quote(2+2)
var qb <- qmacro_block() {
    let foo = $e(expr)
}
print(describe(qb))
```

prints:

```
let foo:auto const = (2 + 2)
```

$b(array-of-expr)

$b takes an array<ExpressionPtr> or das::vector<ExpressionPtr> aka dasvector`smart_ptr`Expression as an argument and is replaced with each expression from the input array in sequential order:

```
var qqblk : array<ExpressionPtr>
for ( i in range(3) ) {
    qqblk |> emplace_new <| qmacro(print("{$v(i)}\n"))
}
var blk <- qmacro_block() {
    $b(qqblk)
}
print(describe(blk))
```

prints:

```
print(string_builder(0, "\n"))
print(string_builder(1, "\n"))
print(string_builder(2, "\n"))
```

$a(arguments)

$a takes an array<ExpressionPtr> or das::vector<ExpressionPtr> aka dasvector`smart_ptr`Expression as an argument and replaces call arguments with each expression from the input array in sequential order:

```
var arguments <- [quote(1+2); quote("foo")]
var blk <- qmacro <| somefunnycall(1,$a(arguments),2)
print(describe(blk))
```

prints:

```
somefunnycall(1,1 + 2,"foo",2)
```

Note how the other arguments of the function are preserved, and multiple arguments can be substituted at the same time.

Arguments can be substituted in the function declaration itself. In that case $a expects array<VariablePtr>:

```
var foo <- [
    new Variable(name:="v1", _type<-qmacro_type(type<int>)),
    new Variable(name:="v2", _type<-qmacro_type(type<float>), init<-qmacro(1.2))
]
var fun <- qmacro_function("show") <| $ ( a: int; $a(foo); b : int ) {
    return a + b
}
print(describe(fun))
```

prints:

```
def public add ( a:int const; var v1:int; var v2:float = 1.2f; b:int const ) : int {
    return a + b
}
```

$t(type)

`$t` takes a `TypeDeclPtr` as an input and substitutes it with the type expression. In the following example:

```
var subtype <- typeinfo ast_typedecl(type<int>)
var blk <- qmacro_block() {
    var a : $t(subtype)?
}
print(describe(blk))
```

we create pointer to a subtype:

```
var a:int? -const
```

$c(call-name)

`$c` takes a `string` or `das_string` as an input, and substitutes the call expression name:

```
var cll = "somefunnycall"
var blk <- qmacro ( $c(cll)(1,2) )
print(describe(blk))
```

prints:

```
somefunnycall(1,2)
```

See also:

Macros for macro registration and the compilation lifecycle where reification is used, *Generic programming* for `typeinfo` and AST manipulation context.

2.35 Built-in Functions

Daslang built-in functions fall into two categories:

- **Intrinsic functions** — compiled directly into the AST as dedicated expression nodes (e.g. `invoke`, `assert`, `debug`). They are documented in this section.
- **Standard library functions** — defined in `builtin.das` and other standard modules that are always available. These include container operations (`push`, `sort`, `find_index`), iterator helpers (`each`, `next`), clone, lock, and serialization functions. They are documented in the standard library reference (see Built-in functions reference).

This page covers both groups, organized by category.

2.35.1 Invocation

invoke(*block_or_function*, *arguments*)

Calls a block, lambda, or function pointer with the provided arguments:

```
def apply(f : block<(x : int) : int>; value : int) : int {
    return invoke(f, value)
}
```

Blocks, lambdas, and function pointers can also be called using regular call syntax. The compiler will expand `f(value)` into `invoke(f, value)`:

```
def apply(f : block<(x : int) : int>; value : int) : int {
    return f(value)     // equivalent to invoke(f, value)
}
```

(see *Functions*, *Blocks*, *Lambdas*).

2.35.2 Assertions

assert(*x*, *str*)

Triggers an application-defined assert if `x` is `false`. `assert` may be removed in release builds, so the expression `x` must have **no side effects** — the compiler will reject it otherwise:

```
assert(index >= 0, "index must be non-negative")
```

verify(*x*, *str*)

Triggers an application-defined assert if `x` is `false`. Unlike `assert`, `verify` is **never removed** from release builds (it generates `DAS_VERIFY` in C++ rather than `DAS_ASSERT`). Additionally, the expression `x` is allowed to have side effects:

```
verify(initialize_system(), "initialization failed")
```

static_assert(*x*, *str*)

Causes a **compile-time** error if `x` is `false`. `x` must be a compile-time constant. `static_assert` expressions are removed from the compiled program:

```
static_assert(typeinfo(is_pod type<Foo>), "Foo must be POD")
```

concept_assert(*x*, *str*)

Similar to `static_assert`, but the error is reported **one level above** in the call stack. This is useful for reporting type-contract violations in generic functions:

```
def sort_array(var a : auto(TT)[]) {
    concept_assert(typeinfo(is_numeric type<TT>), "sort_array requires numeric type
↪")
    sort(a)
}
```

2.35.3 Debug

debug(*x*, *str*)

Prints the string `str` and the value of `x` (similar to `print`), then **returns** `x`. This makes it suitable for debugging inside expressions:

```
let mad = debug(x, "x") * debug(y, "y") + debug(z, "z")
```

print(*str*)

Outputs the string `str` to the standard output. `print` only accepts strings — to print other types, use string interpolation:

```
print("hello\n")        // ok
print("{13}\n")         // ok, integer is interpolated into the string
// print(13)            // error: print expects a string
```

2.35.4 Panic

panic(*str*)

Terminates execution with the given error message. Panic can be caught with a `try`/`recover` block, but unlike C++ exceptions, `panic` is intended for **fatal errors only**. Recovery may have side effects, and not everything on the stack is guaranteed to recover properly:

```
try {
    panic("something went wrong")
} recover {
    print("recovered from panic\n")
}
```

2.35.5 Memory & Type Utilities

addr(*x*)

Returns a pointer to the value `x`. This is an **unsafe** operation:

```
unsafe {
    var a = 42
    var p = addr(a)      // p is int?
}
```

intptr(*p*)

Converts a pointer (raw or smart) to a `uint64` integer value representing its address:

```
let address = intptr(some_ptr)
```

typeinfo(*trait expression*)

Provides compile-time type information about an expression or a `type<T>` argument. Used extensively in generic programming:

```
typeinfo(sizeof type<float3>)       // 12
typeinfo(typename type<int>)        // "int"
typeinfo(is_pod type<int>)          // true
typeinfo(has_field<x> myStruct)     // true if myStruct has field x
```

(see *Generic Programming* for a full list of typeinfo traits).

2.35.6 Array Operations

The following operations are defined in `builtin.das` and are always available.

Resize & Reserve

resize(*var arr : array<T>; new_size : int*)

Resizes the array to `new_size` elements. New elements are zero-initialized.

resize_and_init(*var arr : array<T>; new_size : int*)

Resizes the array and default-initializes all new elements using the element type's default constructor.

resize_no_init(*var arr : array<T>; new_size : int*)

Resizes without initializing new elements. Only valid for POD/raw element types.

reserve(*var arr : array<T>; capacity : int*)

Pre-allocates memory for at least `capacity` elements without changing the array length.

Push & Emplace

push(*var arr : array<T>; value : T [; at : int]*)

Inserts a **copy** of `value` into the array at index `at` (or at the end if `at` is omitted). Also accepts another `array<T>` or a fixed-size `T[]` to push all elements at once:

```
var a : array<int>
push(a, 1)
push(a, 2, 0)          // insert 2 at beginning
push(a, fixed_array(3, 4, 5))   // push three elements
```

push_clone(*var arr : array<T>; value [; at : int]*)

Clones `value` (deep copy) and inserts the clone into the array.

emplace(*var arr : array<T>; var value : T& [; at : int]*)

Move-inserts `value` into the array, zeroing the source. Preferred for types that own resources (e.g., other arrays, tables, smart pointers):

```
var inner : array<int>
push(inner, 1)
var outer : array<array<int>>
emplace(outer, inner)   // inner is now empty
```

emplace_new(*var arr : array<smart_ptr<T>>; var value : smart_ptr<T>*)

Move-inserts a smart pointer into the back of the array.

Remove & Erase

erase(*var arr : array<T>; at : int [; count : int]*)

Removes the element at index `at` (or `count` elements starting at `at`).

erase_if(*var arr : array<T>; blk : block<(element : T) : bool>*)

Removes all elements for which `blk` returns `true`:

```
erase_if(arr) $(x) { return x < 0 }
```

remove_value(var arr : array<T>; value : T) : bool()

Removes the first occurrence of `value`. Returns `true` if an element was removed.

pop(*var arr : array<T>*)

Removes the last element of the array.

Access & Search

back(var arr : array<T>) : T&()

Returns a reference to the last element. Panics if the array is empty.

empty(arr : array<T>) : bool()

Returns `true` if the array has no elements.

length(arr : T[]) : int()

Returns the compile-time dimension of a fixed-size array.

find_index(arr; value : T) : int()

Returns the index of the first element equal to `value`, or `-1` if not found. Works on dynamic arrays, fixed-size arrays, and iterators.

find_index_if(arr; blk : block<(element : T) : bool>) : int()

Returns the index of the first element satisfying `blk`, or `-1` if not found.

has_value(arr; value) : bool()

Returns `true` if any element equals `value`. Works on any iterable.

subarray(arr; r : range) : array<T>()

Returns a new array containing elements in the specified range.

Sorting

sort(*var arr*)

Sorts the array in ascending order using the default < operator:

```
var a <- array(3, 1, 2)
sort(a)     // a is now [1, 2, 3]
```

sort(*var arr; cmp : block<(x, y : T) : bool>*)

Sorts the array using a custom comparator. The comparator should return `true` if `x` should come before `y`:

```
sort(arr) $(a, b) { return a > b }  // descending order
```

Swap

swap(*var a, b : T&*)

Swaps two values using move semantics through a temporary.

2.35.7 Table Operations

Lookup

key_exists(tab : table<K;V>; key : K) : bool()

Returns `true` if `key` exists in the table.

get(*tab : table<K;V>; key : K; blk : block<(value : V&)>*)

Looks up `key` in the table. If found, the table is locked and `blk` is invoked with a reference to the value. Returns `true` if the key was found:

```
get(tab, "key") $(value) {
    print("found: {value}\n")
}
```

get_value(var tab : table<K;V>; key : K) : V()

Returns a **copy** of the value at `key`. Only works for copyable types.

clone_value(var tab : table<K; smart_ptr<V>>; key : K) : smart_ptr<V>()

Clones the smart pointer value at `key`.

Insert & Emplace

insert(*var tab : table<K;V>; key : K; value : V*)

Inserts a key-value pair. For key-only tables (sets), only the key is needed:

```
var seen : table<string>
insert(seen, "hello")
```

insert_clone(*var tab : table<K;V>; key : K; value : V*)

Clones `value` and inserts the clone into the table.

emplace(*var tab : table<K;V>; key : K; var value : V&*)

Move-inserts `value` into the table at `key`.

insert_default(*var tab : table<K;V>; key : K [; value]*)

Inserts `value` (or a default-constructed value) only if `key` does not already exist.

emplace_default(*var tab : table<K;V>; key : K*)

Emplaces a default value only if `key` does not already exist.

Remove & Clear

erase(var tab : table<K;V>; key : K) : bool()

Removes the entry at `key`. Returns `true` if the key was found and removed.

clear(*var tab : table<K;V>*)

Removes all entries from the table.

Iteration

keys(tab : table<K;V>) : iterator<K>()

Returns an iterator over all keys in the table.

values(tab : table<K;V>) : iterator<V&>()

Returns an iterator over all values in the table (mutable or const depending on the table).

empty(tab : table<K;V>) : bool()

Returns `true` if the table has no entries.

2.35.8 Iterator Operations

each(*iterable*)

Creates an iterator from a range, array, fixed-size array, string, or lambda:

```
for (x in each(my_range)) {
    print("{x}\n")
}
```

Overloads exist for `range`, `urange`, `range64`, `urange64`, `string`, `T[]`, `array<T>`, and `lambda`.

each_enum(*tt : T*)

Creates an iterator over all values of an enumeration type.

Deprecated since version Use: the built-in enumeration iteration instead.

next(var it : iterator<T>; var value : T&) : bool()

Advances the iterator and stores the current value in `value`. Returns `false` when the iterator is exhausted.

nothing(var it : iterator<T>) : iterator<T>()

Returns an empty (nil) iterator.

iter_range(container) : range()

Returns `range(0, length(container))` — useful for index-based iteration.

2.35.9 Conversion Functions

to_array(source) : array<T>()

Converts a fixed-size array or iterator to a dynamic `array<T>`:

```
let fixed = fixed_array(1, 2, 3)
var dynamic <- to_array(fixed)
```

to_array_move(var source) : array<T>()

Moves elements from the source into a new dynamic array.

to_table(source) : table<K;V>()

Converts a fixed-size array of tuples to a table<K;V>, or a fixed-size array of keys to a key-only table (set):

```
var tab <- to_table(fixed_array(("one", 1), ("two", 2)))
```

to_table_move(var source) : table<K;V>()

Moves elements from the source into a new table.

2.35.10 Clone

clone(src) : T()

Creates a deep copy of any value. For arrays and tables, all elements are cloned recursively:

```
var a <- [1, 2, 3]
var b := a          // equivalent to: clone(b, a)
```

clone_dim(*var dst; src*)

Clones a fixed-size array into another of the same dimension.

(see *Clone* for full cloning rules).

2.35.11 Lock Operations

lock(*container; blk*)

Locks an array or table, invokes blk with a temporary handled reference, then unlocks. While locked, the container cannot be resized or modified structurally:

```
lock(my_table) $(t) {
    for (key in keys(t)) {
        print("{key}\n")
    }
}
```

lock_forever(var tab : table<K;V>) : table#()

Locks the table permanently and returns a handled (temporary) reference. This is useful for read-only lookup tables.

lock_data(*var arr : array<T>; blk : block<(var p : T?#; s : int)>*)

Locks the array and provides raw pointer access to the underlying data along with its length.

2.35.12 Serialization

binary_save(*obj; blk : block<(data : array<uint8>#)>*)

Serializes a reference-type object to a binary representation and invokes `blk` with the resulting byte array.

binary_load(*var obj; data : array<uint8>*)

Deserializes a reference-type object from binary data.

2.35.13 Smart Pointer

get_ptr(src : smart_ptr<T>) : T?()

Extracts a raw pointer from a smart pointer.

get_const_ptr(src : smart_ptr<T>) : T? const()

Extracts a const raw pointer from a smart pointer.

add_ptr_ref(src : smart_ptr<T>) : smart_ptr<T>()

Increments the reference count and returns a new smart pointer.

2.35.14 Memory Mapping

map_to_array(*data : void?; len : int; blk*)

Maps raw memory to a temporary mutable array view. This is an **unsafe** operation.

map_to_ro_array(*data : void?; len : int; blk*)

Maps raw memory to a temporary read-only array view. This is an **unsafe** operation.

2.35.15 Vector Construction

Helper functions for constructing vector types from individual components:

```
let v2 = float2(1.0, 2.0)
let v3 = float3(1.0, 2.0, 3.0)
let v4 = float4(1.0, 2.0, 3.0, 4.0)

let i2 = int2(1, 2)
let i3 = int3(1, 2, 3)
let i4 = int4(1, 2, 3, 4)

let u2 = uint2(1u, 2u)
let u3 = uint3(1u, 2u, 3u)
let u4 = uint4(1u, 2u, 3u, 4u)
```

These accept any numeric arguments and convert them to the appropriate element type.

2.35.16 Move Helpers

copy_to_local(a) : T()

Copies a value into a local variable (removes const qualifier).

move_to_local(var a : T&) : T()

Moves a value out of a reference into a local variable.

move_to_ref(*var dst : T&; var src : T*)

Moves (or copies for non-ref types) `src` into `dst`.

2.35.17 Miscellaneous

get_command_line_arguments() : array<string>()

Returns the command-line arguments passed to the program.

CHAPTER

THREE

THE RUNTIME

3.1 Context

Daslang environments are organized into contexts. Compiling a Daslang program produces a `Program` object, which can then be simulated into a `Context`.

`Context` consists of

- name and flags
- functions code
- global variables data
- shared global variable data
- stack
- dynamic memory heap
- dynamic string heap
- constant string heap
- runtime debug information
- locks
- miscellaneous lookup infrastructure

In some sense, a `Context` can be viewed as a Daslang virtual machine — the object responsible for executing code and maintaining state. It can also be viewed as an instance of a class whose methods can be accessed when marked with `[export]`.

Function code, the constant string heap, runtime debug information, and shared global variables are shared between cloned contexts. This allows each context instance to maintain a relatively small memory profile.

The stack can optionally be shared between multiple contexts of different types, keeping the memory profile even smaller.

3.1.1 Initialization and shutdown

Throughout its lifetime, a `Context` goes through initialization and shutdown phases. Context initialization is implemented in `Context::runInitScript`, and shutdown is implemented in `Context::runShutdownScript`. These functions are called automatically when a `Context` is created, cloned, or destroyed. Depending on the application and the `CodeOfPolicies`, they may also be called when `Context::restart` or `Context::restartHeaps` is called.

It is initialized in the following order:

1. All global variables are initialized in the order they are declared, per module.
2. All functions tagged with `[init]` are called in the order they are declared, per module, except for specifically ordered ones.
3. All specifically ordered functions tagged with `[init]` are called in the order they appear after topological sort.

The topological sort order for the init functions is specified in the init annotation.

- `tag` attribute specifies that function will appear during the specified pass
- `before` attribute specifies that function will appear before the specified pass
- `after` attribute specifies that function will appear after the specified pass

Consider the following example:

```
[init(before="middle")]
def a {
    order |> push("a")
}
[init(tag="middle")]
def b {
    order |> push("b")
}
[init(tag="middle")]
def c {
    order |> push("c")
}
[init(after="middle")]
def d {
    order |> push("d")
}
```

The functions will execute in the following order:

1. d
2. b or c, in any order
3. a

During shutdown, the context runs all functions marked with `[finalize]` in the order they are declared, per module.

3.1.2 Macro contexts

For each module that contains macros, an individual context is created and initialized. In addition to regular functions, functions tagged with `[macro]` or `[_macro]` are called during initialization.

Functions tagged with `[macro_function]` are excluded from the regular context and only appear in macro contexts.

Unless a macro module is marked as shared, it will be shut down after compilation. Shared macro modules are initialized during their first compilation, and are shut down during the environment shutdown.

3.1.3 Locking

A context contains a `recursive_mutex` and can be locked and unlocked with the `lock_context` or `lock_this_context` RAII blocks. Cross-context calls via `invoke_in_context` automatically lock the target context.

3.1.4 Lookups

Global variables and functions can be looked up by name or by mangled name hash on both Daslang and C++ side.

3.1.5 Memory allocation and garbage collection

Memory allocation strategies for both the string heap and the regular heap are specified in the `CodeOfPolicies` and options.

To allow garbage collection from inside the context, the following options are necessary:

```
options persistent_heap // this one enables garbage-collectable heap
options gc              // this one enables garbage collection for the variables on the␣
↪stack
```

To collect garbage, from the inside of the context:

```
var collect_string_heap = true
var validate_after_collect = false
heap_collect(collect_string_heap, validate_after_collect)
```

To do the same from the C++ side:

```
context->collectHeap(dummy_line_info_ptr, collect_string_heap, validate_after_collect);
```

See also:

Annotations for `[init]`, `[finalize]`, and `[export]` annotations, *Program structure* for the overall compilation and initialization lifecycle, *Locks* for context and container locking details, *Options* for `persistent_heap`, `gc`, and memory allocation policies, *Macros* for macro context initialization.

3.2 Locks

There are several locking mechanisms available in Daslang, designed to ensure the runtime safety of the code.

3.2.1 Context locks

A `Context` can be locked and unlocked via the `lock` and `unlock` functions from the C++ side. When locked, a `Context` cannot be restarted. `tryRestartAndLock` restarts the context if it is not locked, and then locks it regardless. The main reason to lock a context is when data on the heap is accessed externally. Heap collection is safe to perform on a locked context.

3.2.2 Array and Table locks

An `Array` or `Table` can be locked and unlocked explicitly. When locked, they cannot be modified. Calling `resize`, `reserve`, `push`, `emplace`, `erase`, etc. on a locked `Array` will cause a `panic`. Accessing a locked `Table`'s elements via the `[]` operator will also cause a `panic`.

Arrays are locked when iterated over, preventing modification during iteration. The `keys` and `values` iterators lock a `Table` as well. Tables are also locked during `find*` operations.

3.2.3 Array and Table lock checking

Array and table lock checking extends to data structures that internally contain other arrays or tables.

Consider the following example:

```
var a : array < array<int> >
...
for ( b in a[0] ) {
    a |> resize(100500)
}
```

The `resize` operation on the `a` array will cause `panic` because `a[0]` is locked during the iteration. This test, however, can only happen in runtime. The compiler generates custom `resize` code, which verifies locks:

```
def private builtin`resize ( var Arr:array<array<int> aka numT> explicit; newSize:int␣
↪const ) {
    _builtin_verify_locks(Arr)
    __builtin_array_resize(Arr,newSize,24,__context__)
}
```

The `_builtin_verify_locks` function iterates over the provided data and verifies that each `Array` or `Table` is not locked. If any is locked, a `panic` occurs.

Custom operations will only be generated, if the underlying type needs lock checks.

The following operations perform lock checks on data structures:

```
* a <- b
* return <- a
* resize
* reserve
```

(continues on next page)

(continued from previous page)

```
* push
* push_clone
* emplace
* pop
* erase
* clear
* insert
* for the ``Table``
```

Lock checking can be explicitly disabled:

- for a specific `Array` or `Table` by using the `set_verify_array_locks` and `set_verify_table_locks` functions
- for a structure type with the `[skip_field_lock_check]` annotation
- for an entire function with the `[skip_lock_check]` annotation
- for the entire context with `options skip_lock_checks`
- for the entire context with the `set_verify_context_locks` function

See also:

Arrays and *Tables* for the container types that support locking, *Iterators* for iteration patterns that lock containers, *Annotations* for `[skip_lock_check]` and `[skip_field_lock_check]`, *Options* for `skip_lock_checks`, *Contexts* for context-level locking.

3.3 Type Mangling

daslang uses a compact text encoding called **type mangling** to represent types as strings. Mangled names are used internally for function overload resolution, ABI hashing, and debug information. They are also the format accepted by the C integration API (`daScriptC.h`) when binding interop functions.

The mangling is defined by `TypeDecl::getMangledName` and parsed by `MangledNameParser::parseTypeFromMangledName` in `src/ast/ast_typedecl.cpp`.

3.3.1 Primitive types

Each primitive type has a short mnemonic:

daslang	Mangled	Notes
`void`	`v`	
`bool`	`b`	
`int`	`i`	
`int2`	`i2`	SIMD 2-component vector
`int3`	`i3`	
`int4`	`i4`	
`int8`	`i8`	8-bit signed integer
`int16`	`i16`	16-bit signed integer
`int64`	`i64`	64-bit signed integer
`uint`	`u`	
`uint2`	`u2`	
`uint3`	`u3`	
`uint4`	`u4`	
`uint8`	`u8`	
`uint16`	`u16`	
`uint64`	`u64`	
`float`	`f`	
`float2`	`f2`	
`float3`	`f3`	
`float4`	`f4`	
`double`	`d`	
`string`	`s`	
`range`	`r`	
`urange`	`z`	
`range64`	`r64`	
`urange64`	`z64`	

3.3.2 Qualifiers and modifiers

Qualifiers are **prepended** before the base type they modify. For example, `const int` is `Ci` and `const string&` is `C&s`.

Qualifier	Mangled	Meaning
const	`C`	Constant
ref (&)	`&`	Reference
temporary	`#`	Temporary value
implicit	`I`	Implicit type
explicit	`X`	Explicit type match

3.3.3 Pointers

A raw pointer is `?`, followed by an optional smart-pointer marker:

daslang	Mangled
`Foo?`	`?`
`smart_ptr<Foo>`	`?M`
`smart_ptr<Foo>`	`?W` (native smart pointer)

The pointed-to type is encoded as a *first-type* prefix (see *Composite prefixes* below).

3.3.4 Composite prefixes

Some types carry sub-types. These use numbered prefix wrappers:

Prefix	Meaning	Example
`1<T>`	First sub-type (element type)	`array<int>` → `1<i>A`
`2<T>`	Second sub-type (value type for tables)	`table<string;int>` → `1<s>2<i>T`

3.3.5 Container types

daslang	Mangled	Encoding pattern
`array`	`A`	`1<element>A`
`table`	`T`	`1<key>2<value>T`
`iterator`	`G`	`1<element>G`
`tuple`	`U`	`0<types...>U`
`variant`	`V`	`0<types...>V`

For example:

- `array<int>` → `1<i>A`
- `array<float>` → `1<f>A`
- `table<string;int>` → `1<s>2<i>T`
- `iterator<int>` → `1<i>G`

3.3.6 Fixed-size arrays (dim)

Fixed-size dimensions are encoded with square brackets before the base type:

- `int[3]` → `[3]i`
- `float[2][4]` → `[2][4]f`

3.3.7 Callable types

daslang has three callable types, each with its own suffix:

daslang	Mangled	Characteristics
function pointer	`@@`	No capture, cheapest call
lambda	`@`	Heap-allocated, captures variables
block	`$`	Stack-allocated, cannot outlive scope

The argument list uses the `0<args...>` prefix with arguments separated by semicolons. The callable's own **return type** is not encoded in the mangled name — it is determined by the enclosing context.

Examples:

- `function<(a:int; b:float) : string>` → `0<i;f>@@`
- `lambda<(a:int; b:float) : string>` → `0<i;f>@`
- `block<(a:int; b:float) : string>` → `0<i;f>$`
- `block<(a:int) : void>` → `0<i>$`
- `function<() : void>` → `@@` (no args → no `0<>` prefix)

3.3.8 Structures, handled types, and enumerations

Named types use angle-bracket wrappers with a type-class letter:

Type class	Mangled	Example
Structure	`S<name>` or `S<mod::name>`	`S<MyStruct>`
Handled type	`H<name>` or `H<mod::name>`	`H<model>`
Enumeration	`E<name>`	`E<Color>`
Enumeration8	`E8<name>`	8-bit enum
Enumeration16	`E16<name>`	16-bit enum
Enumeration64	`E64<name>`	64-bit enum

When the type belongs to a module, the module name is included: `H<tutorial_07::model>` or `E<math::MathOp>`.

3.3.9 Type aliases

An alias wraps the underlying type:

- `Y<AliasName>type` — e.g. `Y<IntArray>1<i>A` for `typedef IntArray = array<int>`

3.3.10 Named arguments

Named arguments in callable types use the `N<names...>` prefix:

- `N<a;b>0<i;f>$` — a block taking named arguments `a:int` and `b:float`

3.3.11 Bitfields

daslang	Mangled
bitfield	`t`
bitfield8	`t8`
bitfield16	`t16`
bitfield64	`t64`

3.3.12 Special / internal types

These are used internally by the compiler and macro system:

Type	Mangled	Purpose
auto (infer)	`.`	Auto-inferred type
option	`|`	Type option (overload sets)
typeDecl	`D`	`typeinfo` expression type
alias	`L`	Unresolved alias reference
anyArgument	`*`	Wildcard argument
fakeContext	`_c`	Implicit `__context__` parameter
fakeLineInfo	`_l`	Implicit `__lineinfo__` parameter
typeMacro	`^`	Macro-expanded type
aotAlias	`F`	AOT alias marker

3.3.13 Remove-qualifiers (generics)

In generic function signatures, qualifiers can be explicitly removed:

Trait	Mangled	Meaning
remove ref	`-&`	Strip reference
remove const	`-C`	Strip const
remove temp	`-#`	Strip temporary
remove dim	`-[]`	Strip fixed dimensions

3.3.14 Interop function signatures

The C API function `das_module_bind_interop_function` takes a mangled signature string that describes the full function type. The format is:

```
"return_type arg1_type arg2_type ..."
```

Types are separated by **spaces**. The first type is the return type, followed by each argument type. Each type is a complete mangled name.

Examples

```
"v i"                   →  void func(int)
"v s"                   →  void func(string)
"i i i"                 →  int func(int, int)
"f H<Point2D>"          →  float func(Point2D)
"s H<Point2D>"          →  string func(Point2D)
"s 0<i;f>@@ i f"        →  string func(function<(int,float):string>, int, float)
"s 0<i;f>@  i f"        →  string func(lambda<(int,float):string>, int, float)
"s 0<i;f>$  i f"        →  string func(block<(int,float):string>, int, float)
"v 0<i;f>$"             →  void func(block<(int,float)>)
"1<H<model>>? C1<f>A"   →  model? func(const array<float>)
```

3.3.15 Querying mangled names at runtime

From daslang code, mangled names can be obtained with `typeinfo`:

```
options gen2

def my_function(a : int; b : float) : string {
    return "{a} {b}"
}

[export]
def main() {
    // Function mangled name via @@
    print("mangled: {typeinfo(mangled_name @@my_function)}\n")
}
```

This is useful for debugging type signatures and verifying that C-side mangled strings match the daslang-side expectations.

See also:

tutorial_integration_c_binding_types — binding custom types with mangled names

tutorial_integration_c_callbacks — callable type mangling for function pointers, lambdas, and blocks

tutorial_integration_c_unaligned_advanced — unaligned ABI interop functions

Context — runtime context lookups by mangled name hash

CHAPTER

FOUR

EMBEDDING AND INTEGRATION

This section explains how to embed the daslang scripting language into a host application written in C++ or C (and, by extension, any language that can call C functions). It is organized from simple to advanced:

- **Quick Start** — minimal host program, compilation, and evaluation
- **C++ API Reference** — modules, function/type/enum bindings, cast infrastructure
- **C API Reference** — the `daslang/c_api/c_api.h` header for C-only hosts and FFI
- **External Modules** — building and distributing modules outside the main source tree, `.das_module` descriptors, `find_package(DAS)`
- **Advanced Topics** — AOT compilation, standalone contexts, class adapters

For step-by-step walk-throughs with complete, compilable source code, see the *C++ integration tutorials* and *C integration tutorials*.

4.1 Quick Start

This page covers the minimum needed to run a daslang program from a C++ host application. For a complete step-by-step tutorial, see tutorial_integration_cpp_hello_world.

4.1.1 Virtual machine overview

daslang uses a **tree-based interpreter** — the compiled program is a tree of `SimNode` objects, each representing one operation (add, load, call, etc.). Interpretation walks the tree, calling each node's `eval` method.

Key properties of this design:

- **Statically typed, 128-bit words** — every value fits in `vec4f` (an SSE register), so operations are branchless type dispatches at compile time, not runtime.
- **Zero-copy C++ interop** — the in-memory representation matches C++ ABI, so calling between daslang and C++ has no marshalling overhead.
- **Seamless AOT** — the same `SimNode` tree can be walked by the interpreter *or* compiled ahead-of-time into C++, with identical semantics (aot).
- **Value types vs reference types** — types that fit in 128 bits *may* be value types (passed in registers). Whether a type is a reference type depends on its nature (e.g. handled types are always reference types regardless of size), not purely on size.

Execution context

A `Context` is the runtime state for one daslang program: a call stack, two heap allocators (string heap + heap), and a global variable segment.

- **One context per program** — globals, functions, and type info are per-context.
- **Multiple contexts per thread** — you can have multiple contexts in one thread, but only one executes at a time. Contexts can make direct cross-context calls within a thread via `pinvoke`. Contexts are not thread-safe — do not share a context across threads.
- **Cheap reset** — both heaps use bump allocation. Resetting a stateless script is nearly free (pointer reset, no deallocation).
- **Garbage collection** — stop-the-world, triggered manually by calling `context.collectHeap()` or equivalent from the host. There is currently no automatic GC triggered by heap pressure.

4.1.2 Minimal host program

Every C++ host follows the same pattern:

```
#include "daScript/daScript.h"
using namespace das;

void run_script() {
    TextPrinter tout;                          // output sink
    ModuleGroup dummyLibGroup;                  // module group
    auto fAccess = make_smart<FsFileAccess>(); // file system access

    // 1. Compile the script
    auto program = compileDaScript(
        getDasRoot() + "/path/to/script.das",
        fAccess, tout, dummyLibGroup);
    if (program->failed()) {
        for (auto & err : program->errors) {
            tout << reportError(err.at, err.what, err.extra,
                                err.fixme, err.cerr);
        }
        return;
    }

    // 2. Simulate (link + initialize)
    Context ctx(program->getContextStackSize());
    if (!program->simulate(ctx, tout)) {
        for (auto & err : program->errors) {
            tout << reportError(err.at, err.what, err.extra,
                                err.fixme, err.cerr);
        }
        return;
    }

    // 3. Find and call a function
    auto fn = ctx.findFunction("main");
    if (fn) {
```

(continues on next page)

(continued from previous page)

```
        ctx.evalWithCatch(fn, nullptr);
        if (auto ex = ctx.getException()) {
            tout << "Exception: " << ex << "\n";
        }
    }
}

int main(int, char * []) {
    NEED_ALL_DEFAULT_MODULES;     // register built-in modules
    Module::Initialize();         // initialize the module system
    run_script();
    Module::Shutdown();           // clean up
    return 0;
}
```

The three-step flow is always the same:

1. **Compile** — `compileDaScript` parses and type-checks the `.das` file, producing a `ProgramPtr`.
2. **Simulate** — `program->simulate(ctx, tout)` generates the `SimNode` tree and initializes globals.
3. **Evaluate** — `ctx.evalWithCatch(fn, nullptr)` runs a function.

4.1.3 Key types

`TextPrinter`
: Output sink that writes to stdout. Any `TextWriter` subclass works.

`FsFileAccess`
: File system access for the compiler. `setFileInfo` can register virtual files for compiling from strings (tutorial_integration_cpp_dynamic_scripts).

`ModuleGroup`
: Groups modules for shared compilation state. For simple hosts, an empty `dummyLibGroup` suffices.

`Context`
: Runtime state — stack, heaps, globals. Created with the stack size reported by the compiled program.

`SimFunction`
: Pointer to a compiled function. Retrieved via `ctx.findFunction("name")`.

4.1.4 Calling daslang functions with arguments

To pass arguments and receive return values, use `das_invoke_function`:

```
// For: def add(a, b : int) : int
auto fn = ctx.findFunction("add");
int32_t result = das_invoke_function<int32_t>::invoke(
    &ctx, nullptr, fn, 10, 20);
```

This handles argument marshalling automatically and is the preferred approach. See tutorial_integration_cpp_calling_functions for a complete example.

4.1.5 Next steps

- **Bind C++ functions** → tutorial_integration_cpp_binding_functions
- **Bind C++ types** → tutorial_integration_cpp_binding_types
- **Create custom modules** → tutorial_integration_cpp_custom_modules
- **External modules** → embedding_external_modules
- **Full API reference** → embedding_modules

4.2 C++ API Reference

This page is the comprehensive reference for binding C++ code to daslang. For step-by-step tutorials with compilable examples, see the *C++ integration tutorials*.

4.2.1 Creating a module

Derive from `Module`, register bindings in the constructor, and use `REGISTER_MODULE`:

```
class Module_MyMod : public Module {
public:
    Module_MyMod() : Module("my_module_name") {
        ModuleLibrary lib(this);
        lib.addBuiltInModule();
        // addAnnotation, addExtern, addEnumeration, addConstant ...
    }
};
REGISTER_MODULE(Module_MyMod);
```

The host uses `NEED_MODULE(Module_MyMod)` before `Module::Initialize()`. Scripts access it with `require my_module_name`.

See tutorial_integration_cpp_custom_modules for a complete example.

ModuleAotType

Three AOT compilation modes for modules:

no_aot
: Module functions are never AOT-compiled. Used for dynamic or debug-only modules.

hybrid
: Functions use full ABI (slower calls, but no semantic hash dependency). Required when the module may be compiled separately from the script.

cpp
: Full AOT — function calls use semantic hashes for dispatch. Default and fastest mode.

Embedding daslang source in modules

Use the XDD mechanism (`CMAKE_TEXT_XXD` CMake macro) to embed `.das` files as C++ byte arrays, then compile them with `compileBuiltinModule`:

```
#include "my_module.das.inc"

compileBuiltinModule("my_module.das",
    my_module_das, sizeof(my_module_das));
```

See tutorial_integration_cpp_class_adapters for a full example.

Builtin module constants

```
addConstant(*this, "MY_CONST", 42);
```

The type is inferred automatically from the C++ value.

4.2.2 Binding C++ types

To expose a C++ type to daslang, two things are needed:

1. `MAKE_TYPE_FACTORY(DasName, CppType)` at file scope — creates `typeFactory<CppType>` and `typeName<CppType>`
2. A `TypeAnnotation` subclass — describes fields, properties, and behavior

ManagedStructureAnnotation

The most common helper for binding C++ structs (POD or non-POD):

```
struct Object {
    float3   pos;
    float3   vel;
    float    mass;
    float    length() const { return sqrt(pos.x*pos.x + pos.y*pos.y); }
};

MAKE_TYPE_FACTORY(Object, Object)

struct ObjectAnnotation
    : ManagedStructureAnnotation<Object, true /* canNew */> {
    ObjectAnnotation(ModuleLibrary & ml)
        : ManagedStructureAnnotation("Object", ml) {
        addField<DAS_BIND_MANAGED_FIELD(pos)>("pos", "pos");
        addField<DAS_BIND_MANAGED_FIELD(vel)>("vel", "vel");
        addField<DAS_BIND_MANAGED_FIELD(mass)>("mass", "mass");
        addProperty<DAS_BIND_MANAGED_PROP(length)>(
            "length", "length");
    }
};

// In module constructor:
addAnnotation(make_smart<ObjectAnnotation>(lib));
```

Key points:

- **Field binding order matters** — if type B contains type A, register A's annotation first.
- **Properties** look like fields in daslang (`obj.length`) but call C++ methods.
- `addFieldEx` provides custom offsets and type overrides.
- `addPropertyExtConst` handles const/non-const overloads.
- Auto-implements `walk` for data inspection.

Handled types are **reference types** in daslang — mutable local variables (`var`) require `unsafe` blocks. Provide factory functions (`make_xxx()`) returning by value so scripts can use `let x = make_xxx()` without `unsafe`.

See tutorial_integration_cpp_binding_types for a complete example.

DummyTypeAnnotation

Exposes an opaque type with no accessible fields:

```
addAnnotation(make_smart<DummyTypeAnnotation>(
    "OpaqueHandle", "OpaqueHandle", sizeof(OpaqueHandle),
    alignof(OpaqueHandle)));
```

Use case: passing handles through daslang without exposing internals.

ManagedVectorAnnotation

Exposes `std::vector<T>` with automatic `push`, `pop`, `clear`, `resize`, `length`, and iteration support:

```
addAnnotation(make_smart<ManagedVectorAnnotation<int32_t>>(
    "IntVector", lib));
```

ManagedValueAnnotation

For POD types passed by value (must fit in 128 bits). Requires `cast<T>` specialization.

TypeAnnotation virtual methods

For full control, subclass `TypeAnnotation` directly. Key methods:

Capability queries: `canAot`, `canCopy`, `canMove`, `canClone`, `isPod`, `isRawPod`, `isRefType`, `isLocal`, `canNew`, `canDelete`, `canDeletePtr`, `needDelete`, `isIndexable`, `isIterable`, `isShareable`, `isSmart`, `canSubstitute`

Size and alignment: `getSizeOf`, `getAlignOf`

Field access: `makeFieldType`, `makeSafeFieldType`

Indexing and iteration: `makeIndexType`, `makeIteratorType`

Simulation: `simulateDelete`, `simulateDeletePtr`, `simulateCopy`, `simulateClone`, `simulateRef2Value`, `simulateGetNew`, `simulateGetAt`, `simulateGetAtR2V`, `simulateGetIterator`

AOT visitors: `aotPreVisitGetField`, `aotVisitGetField`, `aotPreVisitGetFieldPtr`, `aotVisitGetFieldPtr`

4.2.3 Cast infrastructure

`cast<T>` converts between `vec4f` (daslang's universal register) and C++ types:

```
// Value types (≤128 bits)
vec4f v = cast<int32_t>::from(42);
int32_t i = cast<int32_t>::to(v);

// Reference types (pointer packed into vec4f)
vec4f v = cast<MyObj &>::from(obj);
MyObj & ref = cast<MyObj &>::to(v);
```

`MAKE_TYPE_FACTORY(DasName, CppType)` creates the `typeFactory` and `typeName` specializations needed by the binding system:

```
MAKE_TYPE_FACTORY(Point3, float3)
```

This creates `typeFactory<float3>::make()` (returns `TypeDeclPtr`) and `typeName<float3>()` (returns `"Point3"`).

Type aliases — to expose `Point3` as an alias for `float3`, provide a custom `typeFactory<Point3>` specialization that returns the existing `float3` type declaration.

4.2.4 Binding C++ functions

addExtern + DAS_BIND_FUN

```
int32_t add(int32_t a, int32_t b) { return a + b; }

addExtern<DAS_BIND_FUN(add)>(*this, lib, "add",
    SideEffects::none, "add")
        ->args({"a", "b"});
```

Return by value (> 128 bits) — use `SimNode_ExtFuncCallAndCopyOrMove`:

```
addExtern<DAS_BIND_FUN(make_matrix),
          SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "make_matrix",
    SideEffects::none, "make_matrix");
```

Return by reference — use `SimNode_ExtFuncCallRef`:

```
addExtern<DAS_BIND_FUN(get_ref),
          SimNode_ExtFuncCallRef>(
    *this, lib, "get_ref",
    SideEffects::modifyExternal, "get_ref");
```

See tutorial_integration_cpp_binding_functions for a complete example.

Binding C++ methods

daslang has no member functions — methods are free functions where the first argument is `self`. Pipe syntax (`obj |> method()`) provides method-call ergonomics:

```
using method_get = DAS_CALL_MEMBER(Counter::get);

addExtern<DAS_CALL_METHOD(method_get)>(*this, lib, "get",
    SideEffects::none,
    DAS_CALL_MEMBER_CPP(Counter::get))
        ->args({"self"});
```

- Non-const methods: `SideEffects::modifyArgument`
- Const methods: `SideEffects::none`

See tutorial_integration_cpp_methods for details.

Binding operators

Register functions with the operator symbol as the daslang name:

```
addExtern<DAS_BIND_FUN(vec3_add),
          SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "+",
    SideEffects::none, "vec3_add")
        ->args({"a", "b"});
```

Available operator names: +, -, *, /, %, <<, >>, <, >, <=, >=, &, |, ^.

`addEquNeq<T>(*this, lib)` binds both == and !=.

See tutorial_integration_cpp_operators_and_properties.

`addInterop` — low-level binding

`addInterop` provides raw access to simulation-level arguments (`vec4f *`) and call metadata (`SimNode_CallBase *`). When a template parameter is `vec4f`, it means "any daslang type":

```
vec4f my_interop(Context & ctx, SimNode_CallBase * call,
                 vec4f * args) {
    TypeInfo * ti = call->types[0];  // type of first arg
    // inspect ti->type, ti->structType, etc.
    return v_zero();
}

addInterop<my_interop, void, vec4f>(*this, lib, "my_func",
    SideEffects::none, "my_interop");
```

Key capabilities (vs `addExtern`):

- Access to `call->types[]` — per-argument `TypeInfo`
- Access to `call->debugInfo` — source location of call site
- `vec4f` argument type = "any" — accepts any daslang type

TypeInfo union warning: `TypeInfo` has a union — `structType`, `enumType`, and `annotation_or_name` share memory. Which member is valid depends on `ti->type`:

- `tStructure` → `ti->structType`
- `tEnumeration` / `tEnumeration8` / `tEnumeration16` → `ti->enumType`
- `tHandle` → `ti->getAnnotation()`

See tutorial_integration_cpp_interop.

4.2.5 Binding C++ enumerations

```
// MUST come BEFORE `using namespace das`
DAS_BASE_BIND_ENUM(CppEnum, DasName, Value1, Value2, Value3)

using namespace das;

// In module constructor:
addEnumeration(make_smart<EnumerationDasName>());
```

- `DAS_BASE_BIND_ENUM` creates `EnumerationDasName` class + `typeFactory<CppEnum>`
- `DAS_BASE_BIND_ENUM_98` — for unscoped (C-style) enums
- Place enum macros **before** `using namespace das` to avoid name collisions

See tutorial_integration_cpp_binding_enums.

4.2.6 Function side effects

Every bound function declares its side effects:

SideEffects::none
: Pure function — no external state changes.

SideEffects::unsafe
: Function is unsafe.

SideEffects::userScenario
: User-defined scenario.

SideEffects::modifyExternal
: Modifies external state (stdout, files, globals).

SideEffects::accessExternal
: Reads external state.

SideEffects::modifyArgument
: Mutates a reference argument.

SideEffects::accessGlobal
: Reads shared global state.

SideEffects::invoke
: Calls daslang callbacks (blocks, functions, lambdas).

SideEffects::worstDefault
: Safe fallback — assumes all side effects.

4.2.7 Callbacks — blocks, functions, lambdas

Three closure types exist for calling daslang code from C++:

Type	Template	Invocation	Lifetime
Block	`TBlock<Ret, Args...>`	`das_invoke<Ret>::invoke(ctx, at, blk, args...)`	Stack-bound (current call only)
Function	`TFunc<Ret, Args...>`	`das_invoke_function<Ret>::invoke(ctx, at, fn, args...)`	Context-bound (storable)
Lambda	`TLambda<Ret, Args...>`	`das_invoke_lambda<Ret>::invoke(ctx, at, lmb, args...)`	Heap-allocated (captures variables)

Block callback example:

```
void with_values(int32_t a, int32_t b,
                 const TBlock<void, int32_t, int32_t> & blk,
                 Context * context, LineInfoArg * at) {
    das_invoke<void>::invoke(context, at, blk, a, b);
}

addExtern<DAS_BIND_FUN(with_values)>(*this, lib, "with_values",
    SideEffects::invoke, "with_values")
        ->args({"a", "b", "blk", "context", "at"});
```

Use `SideEffects::invoke` for any function that invokes script callbacks.

See tutorial_integration_cpp_callbacks.

4.2.8 File access

`FsFileAccess` is the default file system access layer. Override `getNewFileInfo` and `getModuleInfo` for custom file resolution:

```
struct ModuleInfo {
    string  moduleName;
    string  fileName;
    string  importDir;
};
```

`setFileInfo` registers virtual files (strings as files):

```
auto fAccess = make_smart<FsFileAccess>();
auto fileInfo = make_unique<TextFileInfo>(
    text.c_str(), uint32_t(text.length()), false);
fAccess->setFileInfo("virtual.das", das::move(fileInfo));
```

See tutorial_integration_cpp_dynamic_scripts.

4.2.9 Project file resolution

A custom `module_get` function resolves `require` paths:

```
options gen2
require daslib/ast_boost public

[function_macro(name="module_get")]
class ModuleGetMacro : AstModuleGetMacro {
    def override getModule(
        req : string;
        from : string
    ) : ModuleInfo {
        // resolve require paths here
    }
}
```

See tutorial_integration_cpp_custom_modules for the C++ equivalent.

4.3 C API Reference

The C API (`daScriptC.h`) provides a pure-C interface for embedding daslang in applications that cannot use C++. It covers the full lifecycle: initialization, compilation, simulation, function calls, type binding, serialization, and sandboxing.

For step-by-step tutorials with compilable examples, see the *C integration tutorials*.

  - *Type aliases*
- *String allocation*
- *Context variables*
- *Serialization*
- *AOT checking*
- *Module groups*
- *Side effects*

4.3.1 When to use the C API

The C API is appropriate when:

- The host application is written in C (not C++).
- You need a stable ABI boundary (e.g. loading daslang as a shared library from another language).
- You want to avoid C++ name mangling in your interop layer.

The C++ API (`daScript.h`) is more ergonomic and exposes more functionality. The C API covers the most common embedding scenarios but does not expose every C++ feature (e.g. class adapters, custom annotations).

4.3.2 Initialization and shutdown

```
#include "daScript/daScriptC.h"

int main() {
    das_initialize();
    // ... use daslang ...
    das_shutdown();
    return 0;
}
```

das_initialize()
Must be called once before any other C API call. Registers all built-in modules.

das_shutdown()
Frees all internal structures. Call once when done.

4.3.3 Text output

```
das_text_writer * tout = das_text_make_printer();  // stdout
das_text_writer * buf  = das_text_make_writer();   // string buffer
das_text_output(tout, "hello\n");
das_text_release(tout);
das_text_release(buf);
```

das_text_make_printer()
Creates a writer that prints to stdout.

das_text_make_writer()
Creates a writer that accumulates output in an internal buffer.

das_text_release(writer)
Frees a text writer.

4.3.4 File access

```
das_file_access * fAccess = das_fileaccess_make_default();
// ... compile scripts ...
das_fileaccess_release(fAccess);
```

das_fileaccess_make_default()
Creates a filesystem-backed file access.

das_fileaccess_make_project(project_file)
Creates a file access backed by a `.das_project` file for sandboxing. See tutorial_integration_c_sandbox.

das_fileaccess_introduce_file(access, name, content, owns)
Registers a virtual file from a C string. If `owns` is non-zero, the content is copied; otherwise the caller must keep it alive.

das_fileaccess_introduce_file_from_disk(access, name, disk_path)
Reads a file from disk and caches it under a virtual path.

das_fileaccess_introduce_daslib(access)
Pre-loads all standard library modules into the cache.

das_fileaccess_introduce_native_modules(access)
Pre-loads all native plugin modules listed in `external_resolve.inc`.

das_fileaccess_lock(access) / das_fileaccess_unlock(access)
While locked, only pre-introduced files can be accessed — filesystem reads are blocked. Essential for sandboxing.

das_get_root(buf, maxbuf)
Copies the daslang root path into `buf`.

4.3.5 Compilation

```
das_module_group * lib = das_modulegroup_make();
das_program * program = das_program_compile(
    "script.das", fAccess, tout, lib);

if (das_program_err_count(program) > 0) {
    for (int i = 0; i < das_program_err_count(program); i++) {
        das_error * err = das_program_get_error(program, i);
        das_error_output(err, tout);
    }
}
```

das_program_compile(file, access, tout, libgroup)
Compiles a `.das` file. Always returns non-NULL; check `das_program_err_count()` for errors.

das_program_compile_policies(file, access, tout, libgroup, policies)
Same as above but applies custom `CodeOfPolicies`.

das_program_release(program)
Frees a compiled program.

das_program_err_count(program)
Returns the number of errors (0 = success).

das_program_context_stack_size(program)
Returns the minimum stack size for simulation.

das_program_get_error(program, index)
Returns the i-th error object.

See tutorial_integration_c_hello_world.

Compilation policies

```
das_policies * pol = das_policies_make();
das_policies_set_bool(pol, DAS_POLICY_AOT, 1);
das_policies_set_bool(pol, DAS_POLICY_NO_UNSAFE, 1);
das_policies_set_int(pol, DAS_POLICY_STACK, 65536);

das_program * prog = das_program_compile_policies(
    "script.das", fAccess, tout, lib, pol);
das_policies_release(pol);
```

Boolean policies (`das_bool_policy`):

Flag	Effect
`DAS_POLICY_AOT`	Enable AOT compilation linking
`DAS_POLICY_NO_UNSAFE`	Forbid `unsafe` blocks
`DAS_POLICY_NO_GLOBAL_VARIABLES`	Forbid module-level `var` declarations
`DAS_POLICY_NO_GLOBAL_HEAP`	Forbid heap allocations for globals
`DAS_POLICY_NO_INIT`	Forbid `[init]` functions
`DAS_POLICY_FAIL_ON_NO_AOT`	Treat missing AOT as an error
`DAS_POLICY_THREADLOCK_CONTEXT`	Enable context mutex for threading
`DAS_POLICY_INTERN_STRINGS`	Use string interning
`DAS_POLICY_PERSISTENT_HEAP`	Persistent heap (no GC between calls)
`DAS_POLICY_MULTIPLE_CONTEXTS`	Context-safe code generation
`DAS_POLICY_STRICT_SMART_POINTERS`	Strict smart pointer rules
`DAS_POLICY_RTTI`	Generate extended RTTI
`DAS_POLICY_NO_OPTIMIZATIONS`	Disable all optimizations

Integer policies (`das_int_policy`):

Field	Effect
`DAS_POLICY_STACK`	Context stack size in bytes
`DAS_POLICY_MAX_HEAP_ALLOCATED`	Max heap allocated (0 = unlimited)
`DAS_POLICY_MAX_STRING_HEAP_ALLOCATED`	Max string heap allocated (0 = unlimited)
`DAS_POLICY_HEAP_SIZE_HINT`	Initial heap size hint
`DAS_POLICY_STRING_HEAP_SIZE_HINT`	Initial string heap size hint

See tutorial_integration_c_sandbox.

4.3.6 Simulation and context

```
das_context * ctx = das_context_make(
    das_program_context_stack_size(program));
das_program_simulate(program, ctx, tout);

das_function * fn = das_context_find_function(ctx, "main");
das_context_eval_with_catch(ctx, fn, NULL);

das_context_release(ctx);
```

das_context_make(stackSize)
Creates an execution context.

das_program_simulate(program, ctx, tout)
Links the program into the context. Returns 1 on success.

das_context_find_function(ctx, name)
Finds an exported function by name.

das_context_release(ctx)
Frees a context.

See tutorial_integration_c_calling_functions.

4.3.7 Function evaluation

Three calling conventions are provided:

Aligned (vec4f)

Arguments and return values use 16-byte aligned `vec4f`:

```
vec4f args[2];
args[0] = das_result_int(10);
args[1] = das_result_int(20);
vec4f result = das_context_eval_with_catch(ctx, fn, args);
int sum = das_argument_int(result);
```

Unaligned (vec4f_unaligned)

No alignment requirement — preferred for plain C:

```
vec4f_unaligned args[2], result;
das_result_int_unaligned(&args[0], 10);
das_result_int_unaligned(&args[1], 20);
das_context_eval_with_catch_unaligned(ctx, fn, args, 2, &result);
int sum = das_argument_int_unaligned(&result);
```

Complex results (cmres)

For functions returning structures or tuples:

```
MyStruct result_buf;
das_context_eval_with_catch_cmres(ctx, fn, args, &result_buf);
```

The caller allocates the buffer; the function writes the result into it.

4.3.8 Argument helpers

Extracting values from `vec4f` (in interop functions):

`das_argument_int(arg)`, `das_argument_uint(arg)`, `das_argument_int64(arg)`, `das_argument_uint64(arg)`, `das_argument_bool(arg)`, `das_argument_float(arg)`, `das_argument_double(arg)`, `das_argument_string(arg)`, `das_argument_ptr(arg)`, `das_argument_function(arg)`, `das_argument_lambda(arg)`, `das_argument_block(arg)`

Packing values into `vec4f` (for returning from interop functions):

`das_result_void()`, `das_result_int(r)`, `das_result_uint(r)`, `das_result_int64(r)`, `das_result_uint64(r)`, `das_result_bool(r)`, `das_result_float(r)`, `das_result_double(r)`, `das_result_string(r)`, `das_result_ptr(r)`, `das_result_function(r)`, `das_result_lambda(r)`, `das_result_block(r)`

Unaligned variants (`_unaligned` suffix) take a `vec4f_unaligned *` argument instead.

4.3.9 Calling lambdas and blocks

Lambdas and blocks use parallel evaluation functions:

```
// Lambda (heap-allocated closure)
das_context_eval_lambda(ctx, lambda, args);
das_context_eval_lambda_unaligned(ctx, lambda, args, nargs, &result);

// Block (stack-allocated closure)
das_context_eval_block(ctx, block, args);
das_context_eval_block_unaligned(ctx, block, args, nargs, &result);
```

Complex-result variants (`_cmres`) are also available.

See tutorial_integration_c_callbacks.

4.3.10 Binding C functions

```
vec4f my_add(das_context * ctx, das_node * node, vec4f * args) {
    int a = das_argument_int(args[0]);
    int b = das_argument_int(args[1]);
    return das_result_int(a + b);
}

das_module * mod = das_module_create("my_module");
das_module_bind_interop_function(mod, lib, my_add,
    "add", "my_add", SIDEEFFECTS_none, " Cii>i");
```

The `args` string uses daslang's type mangling format.

For unaligned calling convention:

```
void my_add_u(das_context * ctx, das_node * node,
              vec4f_unaligned * args, vec4f_unaligned * result) {
    int a = das_argument_int_unaligned(&args[0]);
    int b = das_argument_int_unaligned(&args[1]);
    das_result_int_unaligned(result, a + b);
}

das_module_bind_interop_function_unaligned(mod, lib, my_add_u,
    "add", "my_add_u", SIDEEFFECTS_none, " Cii>i");
```

See tutorial_integration_c_binding_types.

4.3.11 Binding types

Structures

```
das_structure * st = das_structure_make(lib,
    "MyType", "MyType", sizeof(MyType), alignof(MyType));
das_structure_add_field(st, mod, lib,
    "x", "x", offsetof(MyType, x), "f");    // float
das_structure_add_field(st, mod, lib,
    "y", "y", offsetof(MyType, y), "f");    // float
das_module_bind_structure(mod, st);
```

Field types use mangled-name format ("f" = float, "i" = int, "s" = string, etc.).

Enumerations

```
das_enumeration * en = das_enumeration_make(
    "Color", "Color", 0);
das_enumeration_add_value(en, "red",   "Color::red",   0);
das_enumeration_add_value(en, "green", "Color::green", 1);
das_enumeration_add_value(en, "blue",  "Color::blue",  2);
das_module_bind_enumeration(mod, en);
```

Type aliases

```
das_module_bind_alias(mod, lib, "Color", "i");  // alias Color = int
```

4.3.12 String allocation

```
char * s = das_allocate_string(ctx, "hello");
```

Allocates a copy on the context's string heap. The returned pointer is managed by the context — do not free it manually.

4.3.13 Context variables

After simulation, global variables are accessible by name or index:

```
int idx = das_context_find_variable(ctx, "counter");
if (idx >= 0) {
    int * ptr = (int *)das_context_get_variable(ctx, idx);
    *ptr = 42;  // direct memory write
}

int total = das_context_get_total_variables(ctx);
for (int i = 0; i < total; i++) {
    printf("%s: %d bytes\n",
        das_context_get_variable_name(ctx, i),
        das_context_get_variable_size(ctx, i));
}
```

See tutorial_integration_c_context_variables.

4.3.14 Serialization

```
// Serialize
const void * data;
int64_t size;
das_serialized_data * blob = das_program_serialize(
    program, &data, &size);

// Save data/size to file...

// Deserialize (skips parsing and type inference)
das_program * restored = das_program_deserialize(data, size);

// Clean up
das_serialized_data_release(blob);
```

See tutorial_integration_c_serialization.

4.3.15 AOT checking

```
das_function * fn = das_context_find_function(ctx, "test");
if (das_function_is_aot(fn)) {
    printf("Running as native AOT code\n");
}
```

See tutorial_integration_c_aot.

4.3.16 Module groups

```
das_module_group * lib = das_modulegroup_make();
das_module * mod = das_module_create("my_module");
// ... bind functions, types, enums ...
das_modulegroup_add_module(lib, mod);

das_program * prog = das_program_compile(
    "script.das", fAccess, tout, lib);

das_modulegroup_release(lib);
```

4.3.17 Side effects

Side-effect constants for `das_module_bind_interop_function`:

SIDEEFFECTS_none
: Pure function.

SIDEEFFECTS_unsafe
: Function is unsafe.

SIDEEFFECTS_modifyExternal
: Modifies external state (stdout, files).

SIDEEFFECTS_accessExternal
: Reads external state.

SIDEEFFECTS_modifyArgument
: Mutates a reference argument.

SIDEEFFECTS_modifyArgumentAndExternal
: Both modifyArgument and modifyExternal.

SIDEEFFECTS_accessGlobal
: Reads shared global state.

SIDEEFFECTS_invoke
: Calls daslang callbacks.

SIDEEFFECTS_worstDefault
: Safe fallback — assumes all side effects.

4.4 External Modules

This page explains how to create, build, and distribute daslang modules **outside** the main daScript repository. External modules are compiled and distributed separately from the daslang compiler, and can contain C++ bindings, pure daslang code, or both.

For binding C++ functions and types *within* the compiler source tree, see embedding_modules. For step-by-step integration tutorials, see tutorial_integration_cpp_hello_world.

4.4.1 Overview

daslang has two compiler binaries:

Binary	CMake target	Module resolution
`daslang_static`	`daslang_static`	All modules are compiled-in via `NEED_MODULE` / `NATIVE_MODULE` macros and linked statically.
`daslang` (default)	`daslang`	Modules are discovered at runtime through `.das_module` descriptor scripts and `.shared_module` shared libraries.

The **static** binary is self-contained — every module is compiled into the binary, and no external files are needed at runtime (except for daslang source files). It cannot load external modules.

The **dynamic** binary (the default, and the one shipped in the install) discovers modules at startup by scanning the `modules/` directory for `.das_module` descriptor files. This is the binary that supports external modules.

When you install daslang (via `cmake --install`), the installed SDK contains:

- `bin/daslang` — the dynamic compiler binary
- `lib/` — `libDaScriptDyn` shared library (and static `libDaScript`)
- `include/` — C++ headers
- `lib/cmake/DAS/` — CMake package files (`DASConfig.cmake`, targets)
- `daslib/` — standard library `.das` files
- `modules/` — built-in module directories with `.das_module` descriptors and `.shared_module` shared libraries

4.4.2 Module types

There are three types of external modules:

C++ module (with shared library)
Contains C++ code compiled into a `.shared_module` DLL. Functions and types are registered through the daScript C++ or C binding API. The `.das_module` descriptor uses `register_dynamic_module` to load the shared library.

Pure daslang module (no C++ code)
Implemented entirely in `.das` files. The `.das_module` descriptor uses `register_native_path` to map `require` paths to `.das` files on disk.

Mixed module (C++ and daslang)
Contains both a `.shared_module` DLL and `.das` files. The `.das_module` descriptor uses both `register_dynamic_module` and `register_native_path`.

4.4.3 The `.das_module` descriptor

Every module in the `modules/` directory needs a `.das_module` file. This is a daslang script that the dynamic binary executes at startup to discover the module's components.

The file must:

- Be named exactly `.das_module` (dot-prefixed, no other name)
- Live at `modules/<module_name>/.das_module`
- Export an `initialize` function

Basic structure:

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    // registration calls go here
}
```

The `project_path` argument is the absolute path to the module's directory (e.g., `/path/to/modules/myModule`). Use string interpolation to build paths: `"{project_path}/myLib.shared_module"`.

Registration functions

register_dynamic_module(path, class_name)

Loads a `.shared_module` shared library and registers a C++ module class. The `class_name` must match the `REGISTER_MODULE` or `REGISTER_DYN_MODULE` call in the C++ source:

```
register_dynamic_module("{project_path}/myLib.shared_module", "Module_MyMod")
```

Guard with `das_is_dll_build()` — this function only makes sense in the dynamic binary:

```
if (das_is_dll_build()) {
    register_dynamic_module("{project_path}/myLib.shared_module", "Module_MyMod")
}
```

register_native_path(top, from, to)

Maps a `require` path to a `.das` file on disk. When a script says `require foo/bar`:

- `top` = `"foo"` (the module namespace, before the first `/`)
- `from` = `"bar"` (the path after the namespace)
- `to` = absolute path to the `.das` file

```
register_native_path("mymod", "utils", "{project_path}/das/utils.das")
// Now `require mymod/utils` resolves to <project_path>/das/utils.das
```

This function does not need a `das_is_dll_build()` guard.

Complete examples

C++ module (one module class):

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    if (das_is_dll_build()) {
        register_dynamic_module("{project_path}/dasModuleUnitTest.shared_module",
↪"Module_UnitTest")
    }
}
```

Pure daslang module (multiple `.das` files):

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    let paths = ["peg", "meta_ast", "parse_macro", "parser_generator"]
    for (path in paths) {
        register_native_path("peg", "{path}", "{project_path}/peg/{path}.das")
    }
}
```

Mixed module (C++ shared library + daslang files):

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    let daslib_paths = [
        "llvm_boost", "llvm_debug", "llvm_jit", "llvm_targets",
        "llvm_jit_intrin", "llvm_jit_common", "llvm_dll_utils"
    ]
    let bindings_paths = [
        "llvm_const", "llvm_enum", "llvm_func", "llvm_struct"
    ]
    for (path in daslib_paths) {
        register_native_path("llvm", "daslib/{path}", "{project_path}/daslib/{path}.das")
    }
    for (path in bindings_paths) {
        register_native_path("llvm", "bindings/{path}", "{project_path}/bindings/{path}.
↪das")
    }
}
```

Multiple C++ module classes from one shared library:

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    if (das_is_dll_build()) {
        register_dynamic_module("{project_path}/dasModuleImgui.shared_module", "Module_
↪dasIMGUI")
        register_dynamic_module("{project_path}/dasModuleImgui.shared_module", "Module_
↪dasIMGUI_NODE_EDITOR")
        register_dynamic_module("{project_path}/imguiApp.shared_module", "Module_imgui_
↪app")
    }
}
```

4.4.4 Module resolution order

When the compiler encounters a `require foo/bar` statement, it resolves the module through the following chain (in order):

1. **Parse** the require path: `top = "foo"`, `mod_name = "bar"`
2. **daslib** — if `top == "daslib"`, resolve from the `daslib/` directory
3. **Static modules** — try `NATIVE_MODULE` macro matches (compiled into the static binary only)
4. **Dynamic modules** — try `g_dyn_modules_resolve` entries populated by `.das_module` scripts via `register_native_path`
5. **Extra roots** — try the `extraRoots` map (set by the host application)

6. **dastest** — try the `dastest` module name convention

4.4.5 Building a C++ module

A C++ module is a shared library (`.shared_module`) that links against `libDaScriptDyn` and exports a module registration function.

C++ module class

Create a class that inherits from `das::Module` and register functions, types, and enumerations in the constructor:

```
#include "daScript/daScript.h"
#include "daScript/daScriptModule.h"

const char * hello(const char * name, das::Context * ctx, das::LineInfoArg * at) {
    return ctx->allocateString(das::string("Hello, ") + name + "!", at);
}

class Module_Hello : public das::Module {
public:
    Module_Hello() : Module("HelloModule") {
        das::ModuleLibrary lib(this);
        lib.addBuiltInModule();
        das::addExtern<DAS_BIND_FUN(hello)>(*this, lib,
            "hello", das::SideEffects::none, "hello");
    }
};

REGISTER_DYN_MODULE(Module_Hello, Module_Hello);
REGISTER_MODULE(Module_Hello);
```

Note: Use both `REGISTER_DYN_MODULE` and `REGISTER_MODULE`. `REGISTER_DYN_MODULE` provides the exported entry point for the dynamic binary. `REGISTER_MODULE` provides compatibility with the static binary.

C module (using the C API)

For modules written in C (or any language with C FFI), use the C API from `daScript/daScriptC.h`:

```
#include "daScript/daScriptC.h"

vec4f hello_from_c(das_context * ctx, das_node * node, vec4f * args) {
    return das_result_string("Hello from C module!");
}

#ifdef _MSC_VER
    #define EXPORT_API __declspec(dllexport)
#else
    #define EXPORT_API __attribute__((visibility("default")))
#endif
```

(continues on next page)

(continued from previous page)

```
EXPORT_API das_module * register_dyn_Module_Hello() {
    das_module * mod = das_module_create("HelloModule");
    das_module_group * lib = das_modulegroup_make();
    das_module_bind_interop_function(mod, lib, &hello_from_c,
        "hello_from_c", "hello_from_c", SIDEEFFECTS_modifyExternal, "s ");
    das_modulegroup_release(lib);
    return mod;
}
```

Note that C files must not use `extern "C"` wrappers — C linkage is the default in `.c` files.

CMake setup

External modules use `find_package(DAS)` to locate the installed daslang SDK. The SDK exports:

- `DAS::libDaScriptDyn` — the dynamic runtime library
- `DAS::libDaScript` — the static runtime library
- `DAS::daslang` — the compiler binary (for AOT generation)

A minimal `CMakeLists.txt` for a C++ module:

```
cmake_minimum_required(VERSION 3.16)
project(MyModule)

find_package(DAS REQUIRED)

add_library(myModule SHARED my_module.cpp)
target_link_libraries(myModule PRIVATE DAS::libDaScriptDyn)

# Output as .shared_module in the module directory
set_target_properties(myModule PROPERTIES
    LIBRARY_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    RUNTIME_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    SUFFIX ".shared_module"
    PREFIX ""
)
```

The key points are:

- Link against `DAS::libDaScriptDyn` (not `DAS::libDaScript`)
- Set the output suffix to `.shared_module`
- The shared library should be placed in the module directory (next to the `.das_module` descriptor)

AOT for external modules

External C++ modules can use ahead-of-time compilation for daslang helper functions. Use the installed `DAS::daslang` target to run the AOT compiler:

```
add_custom_command(
    OUTPUT "helper.aot.cpp"
    DEPENDS DAS::daslang "helper.das"
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    COMMAND DAS::daslang -aotlib
        "${CMAKE_CURRENT_SOURCE_DIR}/helper.das"
        "${CMAKE_CURRENT_BINARY_DIR}/helper.aot.cpp"
)

add_library(myModule SHARED my_module.cpp helper.aot.cpp)
```

Use `-aotlib` when AOT-compiling a module helper (generates a linkable object). Use `-aot` for standalone scripts.

Note: Always use absolute paths (`CMAKE_CURRENT_SOURCE_DIR` / `CMAKE_CURRENT_BINARY_DIR`) in AOT custom commands. Relative paths fail when the build directory differs from the source directory.

4.4.6 Building a pure daslang module

Pure daslang modules are the simplest — no C++ compilation is needed. The module is just a directory of `.das` files with a `.das_module` descriptor.

Directory layout:

```
modules/
  myModule/
    .das_module
    src/
      utils.das
      parser.das
```

The `.das_module` maps `require` paths:

```
options gen2
require fio

[export]
def initialize(project_path : string) {
    register_native_path("mymod", "utils", "{project_path}/src/utils.das")
    register_native_path("mymod", "parser", "{project_path}/src/parser.das")
}
```

Scripts can then use `require mymod/utils` and `require mymod/parser`.

4.4.7 Using an external module from a host application

Once a module is built and its `.das_module` descriptor is in place, the host application needs to tell the daslang runtime where to find modules.

Dynamic module discovery

The dynamic binary discovers modules automatically from the `modules/` directory relative to the daslang root. Call `require_dynamic_modules` before `Module::Initialize()`:

```
#include "daScript/daScript.h"
#include "daScript/ast/dyn_modules.h"

int main() {
    das::TextPrinter tout;
    auto fAccess = das::make_smart<das::FsFileAccess>();

    NEED_ALL_DEFAULT_MODULES
    das::require_dynamic_modules(fAccess, das::getDasRoot(), "./", tout);
    das::Module::Initialize();

    // ... compile and run scripts as normal
}
```

`require_dynamic_modules` scans every directory under `modules/` for a `.das_module` file and executes it. The `getDasRoot()` function returns the daslang installation root (set via the `DAS_ROOT` environment variable or auto-detected).

4.4.8 Installing external modules

For distribution, install the module directory into the SDK's `modules/` folder.

If your module is part of the main daScript build tree, use the `ADD_MODULE_LIB` / `ADD_MODULE_DAS` CMake macros — they handle both static and dynamic builds automatically.

For standalone modules, add install rules in your `CMakeLists.txt`:

```
# Install .shared_module and .das_module to the SDK
install(TARGETS myModule
    RUNTIME DESTINATION modules/myModule
    LIBRARY DESTINATION modules/myModule
)
install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/.das_module
    DESTINATION modules/myModule
)
# Install any .das files
install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/src/utils.das
    DESTINATION modules/myModule/src
)
```

After installation, the module directory should contain at minimum:

- `.das_module` — the descriptor
- `.shared_module` — the shared library (for C/C++ modules)

- Any `.das` files the module provides

4.4.9 Example: dascript-demo

The dascript-demo repository is a complete, standalone example of using the installed daslang SDK from an external project. It demonstrates:

- Using `find_package(DAS)` to locate the SDK
- Embedding daslang in a C++ host application
- Embedding daslang in a C host application
- Creating external C and C++ modules with `.das_module` descriptors
- AOT compilation for external modules

Repository structure:

```
dascript-demo/
  CMakeLists.txt          # Top-level: find_package(DAS), host executables
  main.cpp                # C++ host - compiles and runs hello.das
  main-C.c                # C host - same, using the C API
  hello.das               # Test script requiring external modules
  modules/
    moduleHello/          # C module example
      .das_module          # Descriptor (register_dynamic_module + register_native_path)
      hello_module.c       # C source - uses daScriptC.h
      hello_module.das     # Pure-das companion module
      CMakeLists.txt       # Builds hello_Module.mod shared library
    moduleHelloCPP/        # C++ module example with AOT
      .das_module          # Descriptor
      hello_module.cpp     # C++ source - Module class + REGISTER_DYN_MODULE
      hello_module.das     # Helper .das file (AOT-compiled into the module)
      CMakeLists.txt       # Builds shared library + AOT custom command
```

The top-level `CMakeLists.txt` is minimal:

```
cmake_minimum_required(VERSION 3.16)
project(das-example)

find_package(DAS REQUIRED)

# C++ host
add_executable(hello_from_das_cpp main.cpp)
target_link_libraries(hello_from_das_cpp PRIVATE DAS::libDaScriptDyn)

# C host
add_executable(hello_from_das_c main-C.c)
target_link_libraries(hello_from_das_c PRIVATE DAS::libDaScriptDyn)
```

The C++ host (`main.cpp`) follows the standard pattern — compile, simulate, evaluate — with the addition of `require_dynamic_modules` to discover external modules:

```
#include "daScript/daScript.h"
#include "daScript/ast/dyn_modules.h"
```

(continues on next page)

(continued from previous page)

```
int main() {
    das::TextPrinter tout;
    auto faccess = das::smart_ptr<das::FsFileAccess>(new das::FsFileAccess);

    NEED_ALL_DEFAULT_MODULES
    das::require_dynamic_modules(faccess, das::getDasRoot(), "./", tout);
    das::Module::Initialize();

    // compile, simulate, evaluate hello.das ...
}
```

The test script (`hello.das`) requires modules from both external module directories:

```
options gen2
require Hello/hello_module
require Hello/hello_module_cpp
require HelloModule
require HelloCPP

[export]
def main() {
    print("Hello world!\n")
    print("{hello_from_das()}\n")       // from das module
    print("{hello_from_c_module()}\n")  // from C module
    print("{hello_from_cpp_module("Daslang")}\n")  // from C++ module
}
```

4.4.10 In-tree modules vs external modules

If your module is part of the main daScript repository (under `modules/`), use the built-in CMake macros:

`ADD_MODULE_LIB(libName, dllName, sources...)`
: Creates both a static library (`libName`) for the static binary and a `.shared_module` DLL (`dllName`) for the dynamic binary. Both are built automatically.

`ADD_MODULE_CPP(ClassName)`
: Registers the module class in `external_need.inc` so the static binary includes it via `NEED_MODULE`.

`ADD_MODULE_DAS(category, subfolder, native)`
: Registers a pure-das module in `external_resolve.inc` for static binary resolution.

These macros handle the dual static/dynamic build automatically. You still need to write a `.das_module` descriptor for the dynamic build.

For modules outside the daScript tree, use `find_package(DAS)` and link against `DAS::libDaScriptDyn` as shown above.

See also:

- embedding_modules — C++ API reference for module bindings
- aot — AOT compilation details
- tutorial_integration_cpp_hello_world — step-by-step C++ host tutorial
- dascript-demo repository — complete external project example

4.5 Advanced Topics

This page covers advanced embedding techniques: ahead-of-time compilation, class adapters, coroutines, dynamic script generation, and standalone contexts.

4.5.1 AOT compilation

daslang can compile scripts ahead-of-time into C++ source files that, when built alongside the host, replace the interpreter with direct native calls. This provides significant performance improvements for hot paths.

AOT pipeline overview

1. **Compile** the `.das` file as normal.
2. **Run** the AOT tool: `daslang -aot script.das output.cpp`
3. **Build** the generated `.cpp` alongside your application and link `libDaScript`.
4. At runtime, `simulate()` detects AOT-compiled functions and uses them instead of the interpreter.

See tutorial_integration_cpp_aot for a step-by-step walkthrough.

AOT annotations

[no_aot]
Prevents AOT compilation for a specific function.

[hybrid]
AOT function uses full ABI — slower calls, but no semantic hash dependency. Required when the module may be compiled separately from the script.

AOT template functions

By default, AOT-generated functions expect blocks to be passed as the C++ `TBlock` class (see *Block*). If the block is small enough, AOT can replace it with a direct call. `setAotTemplate` enables this:

```
addExtern<DAS_BIND_FUN(my_func)>(*this, lib, "my_func",
    SideEffects::invoke, "my_func")
        ->setAotTemplate();
```

This generates inlined block evaluation instead of `das_invoke`.

AOT type customization

Handled types can customize their AOT generation through `TypeAnnotation` virtual methods:

aotPreVisitGetField(writer, fieldName)
Emits code *before* field access (e.g., dereference operator).

aotVisitGetField(writer, fieldName)
Emits the field access itself.

aotPreVisitGetFieldPtr(writer, fieldName)
Same as above, for pointer field access.

aotVisitGetFieldPtr(writer, fieldName)
Emits pointer field access.

Index operations are covered by `das_index<T, IDX>` and `das_iterator<T>` C++ templates. Override these templates to provide AOT-compatible indexing and iteration for custom types.

das_index<T, IDX>
Used for `ExprAt` (`array[index]`) and `ExprSafeAt` (`array?[index]`). Must provide `at(value, index, context)` and `at(value, index, count, context)` methods.

das_iterator<T>
Used for `for` loop iteration. Must provide `main_loop` and associated helper methods.

4.5.2 Class adapters

Class adapters bridge C++ virtual method dispatch to daslang class method implementations. This allows daslang scripts to define classes that a C++ host can polymorphically call through base class pointers.

Pattern overview

1. Define a C++ abstract base class with virtual methods.
2. Generate a "tutorial adapter" class that maps each virtual method to a `das_invoke_function` call.
3. Create a dual-inheritance adapter: `struct Adapter : BaseClass, TutorialAdapter`.
4. In the daslang script, derive a class from the adapter type and override methods.
5. The C++ host calls virtual methods on base class pointers — dispatch reaches daslang automatically.

Key C++ APIs:

- `das_invoke_function<Ret>::invoke(ctx, at, fn, args...)` — calls a daslang function pointer from C++
- `getDasClassMethod(classPtr, offset)` — retrieves the daslang function pointer for a virtual method
- `compileBuiltinModule` — compiles an embedded `.das` module as a built-in
- `class_info(*classPtr)` — gets `StructInfo` for RTTI bridging

The adapter class is typically generated by `log_cpp_class_adapter` from `daslib/cpp_bind`, which produces the boilerplate `get_<method>` / `invoke_<method>` static helpers.

See tutorial_integration_cpp_class_adapters for a complete, compilable example.

4.5.3 Coroutines (generators)

daslang generators (`generator<T>`) can be consumed from C++ through the `Sequence` type. This enables coroutine-style patterns where the host drives execution step by step.

Consuming a generator

1. Call the generator-returning function with `evalWithCatch`, passing a `Sequence *` as the third argument:

```
Sequence it;
ctx.evalWithCatch(fnTest, nullptr, &it);
```

2. Step through values with `builtin_iterator_iterate`:

```
int32_t value = 0;
while (builtin_iterator_iterate(it, &value, &ctx)) {
    // process value
}
```

3. Clean up with `builtin_iterator_close`:

```
builtin_iterator_close(it, &value, &ctx);
```

Even if the generator has finished, always call `close` to release resources. If you break early, `close` is essential.

See tutorial_integration_cpp_coroutines.

4.5.4 Dynamic script generation

Scripts can be constructed as strings and compiled without writing to disk, using virtual files:

```
TextWriter ss;
ss << "options gen2\n";
ss << "var x = 0.0f\n";
ss << "var y = 0.0f\n";
ss << "[export] def eval() : float { return x*x + y*y; }\n";

auto fAccess = make_smart<FsFileAccess>();
auto fileInfo = make_unique<TextFileInfo>(
    ss.str().c_str(), uint32_t(ss.str().length()), false);
fAccess->setFileInfo("virtual.das", das::move(fileInfo));

auto program = compileDaScript("virtual.das", fAccess, tout,
                               dummyLibGroup);
```

After simulation, global variables can be located and written to directly through context memory:

```
int idx = ctx->findVariable("x");
float * xPtr = (float *)ctx->getVariable(idx);
*xPtr = 3.0f;   // set x = 3.0
```

This pattern is useful for expression evaluators, configuration systems, and dynamic code generation.

See tutorial_integration_cpp_dynamic_scripts.

Context variables

ctx.findVariable(name)
: Returns the index of a global variable, or -1 if not found.

ctx.getVariable(index)
: Returns a raw pointer into the context's global data segment.

ctx.getTotalVariables()
: Returns the number of global variables.

ctx.getVariableName(index)
: Returns the variable's name.

ctx.getVariableSize(index)
: Returns the variable's size in bytes.

See tutorial_integration_cpp_context_variables.

4.5.5 Standalone contexts

A standalone context pre-compiles a daslang program into C++ source files (`.das.h` and `.das.cpp`) that embed the entire program state — function tables, variable layouts, and AOT implementations — without requiring runtime compilation.

Benefits:

- **Zero compilation overhead** at runtime — no parsing, no type inference, no simulation.
- **Direct native calls** — exported functions become C++ methods on a `Standalone` class.
- **Deterministic startup** — all code is linked at build time.

Pipeline:

1. Compile-time: `daslang utils/aot/main.das -- -ctx script.das output_dir/`
2. This generates `script.das.h` and `script.das.cpp`.
3. Build both alongside your host application.
4. At runtime, create the `Standalone` context and call functions directly.

```
#include "standalone_ctx_generated/script.das.h"

das::standalone::Standalone ctx;
ctx.test();  // direct call, no findFunction needed
```

See tutorial_integration_cpp_standalone_contexts for a complete example.

4.5.6 Serialization

Compiled programs can be serialized to a binary blob, skipping compilation entirely on subsequent loads:

```
// Save
AstSerializer ser;
program->serialize(ser);
// ser.buffer contains the binary data

// Load
AstSerializer deser;
deser.buffer = saved_data;
auto restored = Program::deserialize(deser);
// simulate and run as normal
```

This is faster than recompilation (skips parsing and type inference) but slower than standalone contexts (still requires simulation).

See tutorial_integration_cpp_serialization and tutorial_integration_c_serialization.

4.5.7 Sandboxing

`CodeOfPolicies` controls what scripts are allowed to do:

- `no_unsafe = true` — forbid `unsafe` blocks
- `no_global_variables = true` — forbid module-level variables
- `no_global_heap = true` — forbid heap allocations for globals
- `no_init = true` — forbid `[init]` functions
- `threadlock_context = true` — enable context mutex

`FsFileAccess` can be locked to restrict file access:

1. Pre-introduce allowed files with `das_fileaccess_introduce_file`.
2. Lock with `das_fileaccess_lock` to prevent filesystem reads.

See tutorial_integration_cpp_sandbox and tutorial_integration_c_sandbox.

4.5.8 File access customization

Override `FsFileAccess` methods for custom file resolution:

getNewFileInfo(fname)
: Resolves file paths and returns file content.

getModuleInfo(req, from)
: Resolves `require` statements to file paths.

setFileInfo(name, info)
: Registers a virtual file from a string — used for dynamic script generation and embedded modules.

For project-level file resolution, implement `module_get` as a function macro (.das-side):

```
options gen2
require daslib/ast_boost public

[function_macro(name="module_get")]
class ModuleGetMacro : AstModuleGetMacro {
    def override getModule(req, from : string) : ModuleInfo {
        // resolve require paths here
    }
}
```

See tutorial_integration_cpp_custom_modules.

CHAPTER

FIVE

TUTORIALS

This section provides hands-on tutorials organized into four groups:

- **Language Tutorials** — learn daslang syntax and standard library features
- **C Integration Tutorials** — embed daslang in a C host using the `daScriptC.h` API
- **C++ Integration Tutorials** — embed daslang in a C++ host using the native `daScript.h` API
- **Macro Tutorials** — write compile-time code transformations using the daslang macro system

5.1 Language Tutorials

These tutorials introduce daslang's core features step by step. Each comes with a companion `.das` file in `tutorials/language/` that you can run directly:

```
daslang.exe tutorials/language/01_hello_world.das
```

The tutorials are designed to be followed in order. Each one builds on concepts introduced in earlier tutorials.

5.1.1 Hello World

Your first daslang program introduces the essential building blocks: `options gen2` to enable modern syntax, `[export] def main` as the entry point, `print` for console output, and string interpolation.

Setup

Every daslang source file in this tutorial series starts with:

```
options gen2
```

This enables gen2 syntax, which requires curly braces `{ }` around blocks and parentheses `( )` around conditions — a familiar style for anyone coming from C, C++, or similar languages.

Entry point

A runnable daslang program needs a `main` function marked with `[export]`:

```
[export]
def main {
    print("Hello, World!\n")
}
```

`[export]` makes the function visible to the host application (in this case the `daslang` interpreter). `print` writes text to the console but does **not** add a newline — you must include `\n` explicitly.

String interpolation

Inside a string literal, any expression enclosed in `{ }` is evaluated and converted to text:

```
let name = "daslang"
print("Welcome to {name}!\n")

let a = 10
let b = 20
print("{a} + {b} = {a + b}\n")
```

To print a literal brace, escape it with a backslash: `\{` and `\}`.

Running the tutorial

```
daslang.exe tutorials/language/01_hello_world.das
```

Expected output:

```
Hello, World!
Welcome to daslang!
2 + 3 = 5
10 + 20 = 30
Use {braces} for interpolation
```

Full source: `tutorials/language/01_hello_world.das`

See also

- **Next:** *Variables and Types* — variables and types
- *Program structure* — how daslang programs are organized
- *Expressions* — expression syntax including string builders

5.1.2 Variables and Types

This tutorial covers mutable and immutable variables, daslang's basic types, type inference, explicit annotations, and the strict no-implicit-conversion rule.

var vs let

Use `var` for mutable variables and `let` for immutable ones:

```
var score = 0
score = 100        // OK - score is mutable

let maxScore = 999
// maxScore = 0    // ERROR - cannot modify a constant
```

Type inference

The compiler infers the type from the right-hand side:

```
var i = 42          // int
var f = 3.14        // float
var b = true        // bool
var s = "hello"     // string
```

You can also state the type explicitly with `: Type`:

```
var x : int = 10
var y : float = 2.5
var z : double = 1.0lf     // 'lf' suffix for double literals
var flag : bool = false
```

No implicit conversions

daslang is strict — you cannot mix `int` and `float` in arithmetic. Both sides must match:

```
var i = 42
var f = 3.14
// let bad = i + f       // ERROR: int + float
let result = float(i) + f    // OK - explicit cast
```

Likewise, `bool` is not interchangeable with `int`:

```
// let wrong = bool(1)   // ERROR
let right = 1 != 0       // use comparison instead
```

Hex literals and unsigned types

Hex literals like `0xFF` are `uint` by default. Use `int(0xFF)` to get an `int`. 64-bit literals use the `l` and `ul` suffixes:

```
let h = 0xFF                  // uint
let hi = int(0xFF)            // int
var bigInt : int64 = 9_000_000_000l
var bigUint : uint64 = 18_000_000_000ul
```

Zero initialization

Variables declared with a type but no initializer are zero-initialized:

```
var zeroInt : int        // 0
var zeroFloat : float    // 0.0
var zeroBool : bool      // false
var zeroString : string  // ""
```

Running the tutorial

```
daslang.exe tutorials/language/02_variables.das
```

Full source: `tutorials/language/02_variables.das`

See also

- **Next:** *Operators* — operators and expressions
- *Datatypes* — complete type reference
- *Constants and enumerations* — constant values

5.1.3 Operators

This tutorial covers arithmetic, comparison, logical, bitwise, and assignment operators, plus the ternary conditional and the distinction between copy (=) and move (<-).

Arithmetic

Both operands must be the same type — no implicit promotion:

```
let a = 17
let b = 5
print("a + b = {a + b}\n")   // 22
print("a / b = {a / b}\n")   // 3  (integer division)
print("a % b = {a % b}\n")   // 2  (remainder)

let x = 10.0
let y = 3.0
print("x / y = {x / y}\n")   // 3.3333333
```

Comparison and logical

```
print("5 == 5 is {5 == 5}\n")       // true
print("5 != 3 is {5 != 3}\n")       // true

let t = true
let f = false
print("true && false = {t && f}\n") // false
print("true || false = {t || f}\n") // true
print("!true = {!t}\n")             // false
```

Logical && and || short-circuit: the right side is not evaluated if the left side already determines the result.

Ternary operator

```
let val = 42
let desc = val > 0 ? "positive" : "non-positive"
print("{val} is {desc}\n")
```

Increment, decrement, and compound assignment

```
var counter = 10
counter++              // 11
counter--              // 10

var n = 100
n += 10                // 110
n -= 20                // 90
n *= 2                 // 180
n /= 3                 // 60
n %= 7                 // 4
```

Bitwise operators

```
let m = 0x0F
let k = 0xF0
print("0x0F & 0xF0 = {int(m & k)}\n")    // 0
print("0x0F | 0xF0 = {int(m | k)}\n")    // 255
print("0x0F ^ 0xF0 = {int(m ^ k)}\n")    // 255
print("0x0F << 4 = {int(m << 4u)}\n")     // 240
```

Copy vs move — introduction

= is copy assignment — works for simple types and copyable values. <- is move assignment — transfers ownership and zeroes the source:

```
var p = 10
var q = p       // q is a copy of p
q = 99          // p is unchanged

var s = 42
var r <- s      // r gets 42, s is zeroed
```

Move matters for arrays, tables, lambdas, and other heap-allocated types. See *Arrays* and *Move, copy, clone*.

Running the tutorial

```
daslang.exe tutorials/language/03_operators.das
```

Full source: `tutorials/language/03_operators.das`

See also

- **Next:** *Control Flow* — control flow
- *Expressions* — full expression reference

5.1.4 Control Flow

This tutorial covers `if`/`elif`/`else`, `while` loops, `for` loops with ranges, `break`, `continue`, and iterating over multiple sequences in parallel.

if / elif / else

In gen2 syntax, parentheses around the condition and braces around the body are required:

```
let temperature = 25
if (temperature > 30) {
    print("It's hot!\n")
} elif (temperature > 20) {
    print("It's pleasant.\n")
} elif (temperature > 10) {
    print("It's cool.\n")
} else {
    print("It's cold!\n")
}
```

while loop

```
var count = 5
while (count > 0) {
    print("countdown: {count}\n")
    count--
}
```

for loop with range

`range(start, end)` iterates from *start* to *end-1*. The `..` operator is shorthand: `0..5` is the same as `range(0, 5)`:

```
for (i in range(0, 5)) {
    print("{i} ")
}
// 0 1 2 3 4

for (i in 0..5) {
    print("{i} ")
}
// 0 1 2 3 4
```

Nested loops

```
for (row in 1..4) {
    for (col in 1..4) {
        let product = row * col
        print("{product}\t")
    }
    print("\n")
}
```

break and continue

`break` exits the innermost loop. `continue` skips to the next iteration:

```
for (i in 0..10) {
    if (i == 3) {
        break
    }
    print("{i} ")
}
// 0 1 2

for (i in 0..8) {
    if (i % 2 != 0) {
        continue
    }
    print("{i} ")
```

(continues on next page)

(continued from previous page)

```
}
// 0 2 4 6
```

Multiple iterators

`for` can iterate over multiple sequences in parallel. The loop stops when the shortest sequence ends:

```
let names = fixed_array("Alice", "Bob", "Carol")
let scores = fixed_array(95, 87, 92)
for (name, score in names, scores) {
    print("{name}: {score}\n")
}
```

Pair a finite sequence with `count()` to get an index:

```
let fruits = fixed_array("apple", "banana", "cherry")
for (idx, fruit in count(), fruits) {
    print("  [{idx}] {fruit}\n")
}
```

`count()` is a built-in infinite iterator that produces 0, 1, 2, … You can also pass a start and step: `count(10, 2)` produces 10, 12, 14, …

Running the tutorial

```
daslang.exe tutorials/language/04_control_flow.das
```

Full source: `tutorials/language/04_control_flow.das`

See also

- **Next:** *Functions* — functions
- *Statements* — full statement reference
- *Iterators* — iterator mechanics

5.1.5 Functions

This tutorial covers defining functions, parameter passing, default arguments, function overloading, returning multiple values with tuples, and early return.

Defining functions

Use `def` to declare a function. Specify parameter types after a colon and the return type after the parameter list:

```
def greet(name : string) {
    print("Hello, {name}!\n")
}

def add(a, b : int) : int {
    return a + b
}
```

When multiple parameters share a type, list them with commas before the single type annotation: `a, b : int`.

Pass-by-value vs pass-by-reference

By default, parameters are passed by value and are immutable. Use `var` and `&` for a mutable reference:

```
def increment(var x : int&) {
    x++
}

var val = 42
increment(val)
print("{val}\n")     // 43
```

Default arguments

Parameters can have default values. Callers may omit them:

```
def formatNumber(value : int; prefix : string = "#") : string {
    return "{prefix}{value}"
}

formatNumber(42)                    // uses default: "#42"
formatNumber(42, [prefix = ">"])    // named argument: ">42"
```

To specify a parameter by name, use named argument syntax: `[name = value]`.

Function overloading

Multiple functions can share the same name if their parameter types differ. The compiler selects the correct overload at compile time:

```
def describe(x : int)    { print("integer: {x}\n") }
def describe(x : float)  { print("float: {x}\n") }
def describe(x : string) { print("string: \"{x}\"\n") }
```

Returning multiple values

Use a named tuple to return multiple values:

```
def divmod(a, b : int) : tuple<quotient:int; remainder:int> {
    return (quotient = a / b, remainder = a % b)
}

let result = divmod(17, 5)
print("{result.quotient}\n")     // 3

// Destructure the tuple
let (q, r) = divmod(23, 7)
```

Early return

Functions can return early from any point:

```
def classifyAge(age : int) : string {
    if (age < 0)  { return "invalid" }
    if (age < 13) { return "child" }
    if (age < 20) { return "teenager" }
    return "adult"
}
```

Running the tutorial

```
daslang.exe tutorials/language/05_functions.das
```

Full source: `tutorials/language/05_functions.das`

See also

- **Next:** *Arrays* — arrays
- *Functions* — full function reference
- *Blocks* — anonymous function blocks
- *Lambdas* — closures and captures

5.1.6 Arrays

This tutorial covers fixed-size arrays, dynamic arrays, `push`/`erase`, iteration, array comprehensions, and move semantics.

Fixed-size arrays

Fixed arrays are value types that live on the stack. Use `fixed_array` to create one:

```
var scores = fixed_array(10, 20, 30, 40, 50)
scores[2] = 99      // modify in-place
```

You can also declare with an explicit type and size:

```
var grid : int[3]
grid[0] = 1
```

Dynamic arrays

Dynamic arrays are heap-allocated and can grow. They use move semantics — they cannot be copied with =:

```
var numbers : array<int>
push(numbers, 10)
push(numbers, 20)
numbers |> push(30)     // pipe syntax
```

Create a dynamic array from a literal with <-:

```
var fruits <- ["apple", "banana", "cherry"]
```

Note the gen2 syntax: square brackets and commas — `["a", "b", "c"]`.

Operations

```
erase(fruits, 1)          // remove element at index 1
print("{length(fruits)}\n")

resize(buf, 5)            // resize to 5 elements (zero-filled)
clear(buf)                // remove all elements
```

Iteration

Use `for` to iterate, and pair with `count()` for an index:

```
for (i, color in count(), colors) {
    print("[{i}] {color}\n")
}
```

Move semantics

Dynamic arrays cannot be copied — they must be moved with `<-`. After a move the source is empty:

```
var source <- [1, 2, 3]
var dest <- source          // dest gets the data
print("{length(source)}\n") // 0
```

Array comprehensions

Build a new array from an expression:

```
var squares <- [for (x in 0..6); x * x]
// [0, 1, 4, 9, 16, 25]
```

Add a filter with `where`:

```
var evens <- [for (x in 0..10); x; where x % 2 == 0]
// [0, 2, 4, 6, 8]
```

Running the tutorial

```
daslang.exe tutorials/language/06_arrays.das
```

Full source: `tutorials/language/06_arrays.das`

See also

- **Next:** *Strings* — strings
- *Arrays* — full array reference
- *Comprehensions* — comprehension syntax
- *Move, copy, clone* — ownership semantics

5.1.7 Strings

This tutorial covers string literals, escape sequences, string interpolation, character access, slicing, type conversions, case conversion, searching, padding, trimming, formatted output, and utilities from the `strings` and `daslib/strings_boost` modules.

String literals

String literals are enclosed in double quotes. Standard escape sequences are supported:

```
print("tab:\there\n")
print("newline:\nhere\n")
print("backslash: \\\n")
print("quote: \"\n")
```

String interpolation

Expressions inside `{ }` are evaluated and inserted:

```
let name = "daslang"
let version = 0.6
print("{name} version {version}\n")
```

Length and character access

`length` returns the number of bytes. `character_at` returns the integer character code at a given index. Convert back with `to_char`:

```
let text = "Hello"
print("{length(text)}\n")     // 5

let ch = character_at(text, 0)
print("{ch}\n")               // 72  (ASCII for 'H')
print("{to_char(ch)}\n")      // H
```

Slicing

`slice` extracts a substring by start and end indices:

```
let phrase = "Hello, World!"
print("{slice(phrase, 0, 5)}\n")   // Hello
print("{slice(phrase, 7)}\n")      // World!
```

Comparison

Strings support the usual comparison operators:

```
"abc" == "abc"   // true
"abc" < "xyz"    // true
```

Use `compare_ignore_case` for case-insensitive comparison. It returns 0 when strings are equal ignoring case:

```
compare_ignore_case("Hello", "hello")   // 0 (equal)
compare_ignore_case("abc", "XYZ")       // negative (a < X)
```

Case conversion

`to_upper` and `to_lower` create new strings with changed case. `capitalize` uppercases the first character and lowercases the rest:

```
to_upper("hello")              // "HELLO"
to_lower("WORLD")              // "world"
capitalize("hello world")      // "Hello world"
```

Converting to and from strings

`string(value)` converts numeric types to a string. For booleans, use string interpolation:

```
let numStr = string(42)      // "42"
let fltStr = string(3.14)    // "3.14"

// For booleans, use interpolation:
let flag = true
print("{flag}\n")            // true
```

To convert strings to numbers, `require strings` and use `to_int` / `to_float`:

```
require strings

let n = to_int("123")       // 123
let f = to_float("3.14")    // 3.14
```

Note: `int("123")` does **not** work — you must use `to_int`.

Searching strings

`find` returns the index of the first occurrence, or `-1` if not found. `contains` (from `daslib/strings_boost`) returns a boolean. `count` counts non-overlapping occurrences. `last_index_of` finds the last occurrence:

```
let sentence = "the quick brown fox"
find(sentence, "quick")            // 4
find(sentence, "slow")             // -1

contains(sentence, "quick")        // true
contains(sentence, "slow")         // false

count("abcabcabc", "abc")          // 3

last_index_of("one/two/three", "/")  // 7
```

Prefix, suffix, and trimming

`starts_with` and `ends_with` test for prefixes and suffixes. `trim_prefix` and `trim_suffix` remove them if present. `strip` removes leading and trailing whitespace:

```
starts_with("image.png", "image")           // true
ends_with("image.png", ".png")              // true

trim_prefix("Hello World", "Hello ")        // "World"
trim_suffix("file.txt", ".txt")             // "file"
trim_suffix("file.txt", ".png")             // "file.txt" (unchanged)

strip("  hello  ")                          // "hello"
```

`is_null_or_whitespace` checks if a string is empty or contains only whitespace:

```
is_null_or_whitespace("")     // true
is_null_or_whitespace("  ")   // true
is_null_or_whitespace("hi")   // false
```

Padding

`pad_left` and `pad_right` pad a string to a minimum width with a fill character (default is space):

```
pad_left("42", 6)          // "    42"
pad_left("42", 6, '0')    // "000042"
pad_right("hi", 6)        // "hi    "
pad_right("hi", 6, '.')   // "hi...."
```

Join and split

`join` concatenates an array of strings with a separator. `split` splits a string by a delimiter:

```
let parts = fixed_array("one", "two", "three")
join(parts, ", ")                    // "one, two, three"

var items <- split("apple,banana,cherry", ",")
// ["apple", "banana", "cherry"]

replace("aabbcc", "bb", "XX")       // "aaXXcc"
```

Formatted output with fmt

`fmt` uses `{fmt}`-library style format specifiers (not `printf`-style). It returns a formatted string:

```
fmt(":d", 42)          // "42"
fmt(":05d", 7)         // "00007"
fmt(":x", 255)         // "ff"
fmt(":.2f", 3.14159)   // "3.14"
```

Building strings

`build_string` constructs strings efficiently using a writer:

```
let result = build_string() $(var writer) {
    write(writer, "Name: ")
    write(writer, "Alice")
    write(writer, ", Score: ")
    write(writer, string(100))
}
print("{result}\n")   // "Name: Alice, Score: 100"
```

Running the tutorial

```
daslang.exe tutorials/language/07_strings.das
```

See also:

Full source: `tutorials/language/07_strings.das`

Next tutorial: *Structs*

String Builder in the language reference

5.1.8 Structs

This tutorial covers declaring structs, field defaults, gen2 construction syntax, nested structs, heap allocation with `new`, and the `with` block.

Declaring a struct

A struct defines a named collection of typed fields. Fields can have default values:

```
struct Point {
    x : float = 0.0
    y : float = 0.0
}

struct Color {
    r : uint8
    g : uint8
    b : uint8
    a : uint8 = uint8(255)
}
```

Construction

In gen2 syntax, construct structs with `StructName(field=value, ...)`:

```
var p1 = Point()                // all defaults
var p2 = Point(x=3.0, y=4.0)   // named fields
```

Partial initialization fills unspecified fields with their defaults:

```
struct Config {
    width : int = 800
    height : int = 600
    title : string = "My App"
    fullscreen : bool = false
}

var cfg = Config(title="Tutorial")
// width=800, height=600, fullscreen=false
```

Structs are pure data

daslang structs have no methods. Write free functions that take a struct as a parameter:

```
def area(r : Rect) : float {
    let w = r.bottomRight.x - r.topLeft.x
    let h = r.bottomRight.y - r.topLeft.y
    return w * h
}
```

Pass by mutable reference with `var` and `&` to modify a struct in place:

```
def translate(var p : Point&; dx, dy : float) {
    p.x += dx
    p.y += dy
}
```

Nested structs

Struct fields can be other structs:

```
struct Rect {
    topLeft : Point = Point()
    bottomRight : Point = Point()
}
```

The `with` block

Inside a `with` block, you can access struct fields without repeating the variable name:

```
var hero = Point()
with (hero) {
    x = 100.0
    y = 200.0
}
```

This is especially convenient for structs with many fields.

Heap allocation

`new` creates a heap-allocated struct and returns a pointer (`Point?`):

```
var p3 = new Point(x=7.0, y=8.0)
print("({p3.x}, {p3.y})\n")

// p3 will be garbage collected when no longer referenced.
// For explicit cleanup, see the tutorial on delete and inscope.
```

Running the tutorial

```
daslang.exe tutorials/language/08_structs.das
```

Full source: `tutorials/language/08_structs.das`

See also

- **Next:** *Enumerations and Bitfields* — enumerations and bitfields
- *Structs* — full struct reference
- *Classes* — structs with virtual methods

5.1.9 Enumerations and Bitfields

This tutorial covers enum declarations, dot-syntax access, underlying types, iterating enums, string conversion, and bitfield flag operations.

Declaring an enum

Enum values are auto-numbered starting from 0:

```
enum Color {
    Red         // 0
    Green       // 1
    Blue        // 2
    Yellow      // 3
}
```

You can assign explicit values:

```
enum HttpStatus {
    OK = 200
    NotFound = 404
    ServerError = 500
}
```

And specify the underlying storage type:

```
enum Direction : uint8 {
    North
    South
    East
    West
}
```

Using enums

Access values with dot syntax — `EnumName.Value`:

```
let favorite = Color.Blue
print("{favorite}\n")          // Blue

if (favorite == Color.Blue) {
    print("it's blue!\n")
}
```

Cast to the underlying integer with `int()`:

```
print("{int(Color.Blue)}\n")   // 2
```

Iterating over enum values

Use `each()` from `daslib/enum_trait` to iterate over all values of an enum. Pass any value of the enum type:

```
require daslib/enum_trait

for (c in each(Color.Red)) {
    print("{c} ")
}
// Red Green Blue Yellow
```

Get the number of values with `typeinfo`:

```
let n = typeinfo enum_length(type<Color>)
```

String conversion

`string(enumValue)` returns the value name. `to_enum` converts a string back:

```
string(Color.Green)                                 // "Green"
to_enum(type<Color>, "Blue")                        // Color.Blue
to_enum(type<Color>, "Purple", Color.Red)           // Color.Red (fallback)
```

Bitfields

Bitfields pack boolean flags into a single integer. Declare them like enums but with `bitfield`:

```
bitfield FilePermissions {
    read
    write
    execute
    hidden
}
```

Set and test flags with dot syntax:

```
var perms : FilePermissions
perms.read = true
perms.write = true

if (perms.read) {
    print("has read\n")
}
```

You can also use `|=` with `EnumName.Value` to set flags:

```
var rwx : FilePermissions
rwx |= FilePermissions.read
rwx |= FilePermissions.write
rwx |= FilePermissions.execute
print("{rwx}\n")               // (read|write|execute)
```

Clear with dot syntax or toggle with `^=`:

```
var flags : FilePermissions = rwx
flags.write = false               // clear
flags ^= FilePermissions.execute  // toggle
```

bitfield_boost

`require daslib/bitfield_boost` adds index-based access for dynamic bit manipulation:

```
flags[0]          // read bit 0 (bool)
flags[1] = true   // set bit 1
each(flags)       // iterate over all bits as bool values
```

This is useful when you need to manipulate bits by index rather than by name.

Running the tutorial

```
daslang.exe tutorials/language/09_enumerations.das
```

Full source: `tutorials/language/09_enumerations.das`

See also

- **Next:** *Tables* — tables (dictionaries)
- *Constants and Enumerations* — full enum reference
- *Bitfields* — bitfield reference

5.1.10 Tables

This tutorial covers declaring tables, inserting and erasing entries, safe lookup with `?[]`, block-based access with `get()`, iteration, table comprehensions, and key-only tables (sets).

Declaring a table

A table maps keys to values. Note the **semicolon** between key and value types:

```
var ages : table<string; int>
```

Inserting values

Use `insert` with pipe syntax:

```
ages |> insert("Alice", 30)
ages |> insert("Bob", 25)
ages |> insert("Carol", 35)
```

Inline construction uses `{ key => value }` with commas:

```
var scores <- { "math" => 95, "english" => 87, "science" => 92 }
```

Safe lookup

`?[]` returns the value if the key exists, falling back to the value after `??`. It does **not** insert missing keys:

```
let age = ages?["Alice"] ?? -1     // 30
let unknown = ages?["Dave"] ?? -1  // -1
```

Use `key_exists` to test for a key:

```
key_exists(ages, "Alice")    // true
key_exists(ages, "Dave")     // false
```

Note: The plain `[]` operator on a table **inserts** a default entry if the key is missing and requires `unsafe`. Prefer `?[]` or `get()` for safe lookups.

Block-based lookup

`get()` runs a block only if the key is found:

```
get(ages, "Bob") $(val) {
    print("Bob is {val} years old\n")
}
```

Pass `var` in the block parameter to modify the value in-place:

```
get(scores, "math") $(var val) {
    val = 100
}
```

Erasing

```
ages |> erase("Bob")
```

Iteration

Use `keys()` and `values()` as parallel iterators:

```
for (subject, score in keys(scores), values(scores)) {
    total += score
}
```

Note: Tables are hash-based — **iteration order is not guaranteed**.

Table comprehensions

Build a table from a `for` expression:

```
var squareMap <- { for (x in 1..6); x => x * x }
```

Key-only tables (sets)

A `table<KeyType>` with no value type acts like a set:

```
var seen : table<string>
seen |> insert("apple")
seen |> insert("banana")
seen |> insert("apple")    // no effect - already present

key_exists(seen, "apple")  // true
```

Safety notes

1. `tab[key]` **inserts** a default value when the key is missing. It requires `unsafe`. Use `?[]` or `get()` instead.
2. Never write `tab[k1] = tab[k2]` — the first `[]` may resize the table, invalidating the reference from the second `[]`.
3. Tables cannot be copied with = — use <- (move) or `:=` (clone).

Running the tutorial

```
daslang.exe tutorials/language/10_tables.das
```

Full source: `tutorials/language/10_tables.das`

See also

- *Tables* — full table reference
- *Comprehensions* — comprehension syntax

Next tutorial: *Tuples and Variants*

5.1.11 Tuples and Variants

This tutorial covers tuples (anonymous and named), the => tuple construction operator, tuple destructuring, variants (tagged unions), and type-safe variant access with `is`, `as`, and `?as`.

Tuples

A tuple groups values of different types into a single value. Access fields by index:

```
var pair = (42, "hello")
print("{pair._0}, {pair._1}\n")    // 42, hello
```

Use `tuple()` for explicit construction:

```
var t = tuple(1, 2.0, "three")
```

The => operator

The => operator creates a 2-element tuple from its left and right operands. It works in any expression context, not just table literals:

```
var kv = "age" => 25       // tuple<string; int>
print("{kv._0}: {kv._1}\n")  // age: 25
```

This is useful for building arrays of key-value pairs:

```
var entries <- ["one" => 1, "two" => 2]
for (entry in entries) {
    print("{entry._0} => {entry._1}\n")
}
```

Table literals also use => — each `key => value` pair forms a tuple that is inserted into the table.

Named tuples

Give a tuple named fields for readability:

```
tuple Point2D {
    x : float
    y : float
}

var p = Point2D(x=3.0, y=4.0)
print("{p.x}, {p.y}\n")
```

Destructuring

Unpack a tuple into individual variables:

```
var (a, b, c) = tuple(10, 20.0, "thirty")
```

This also works in `for` loops:

```
var pairs <- [tuple(1, "one"), tuple(2, "two"), tuple(3, "three")]
for ((num, name) in pairs) {
    print("{num}: {name}\n")
}
```

Variants

A variant holds exactly one of several typed alternatives — a tagged union:

```
variant Value {
    i : int
    f : float
    s : string
}
```

(continues on next page)

(continued from previous page)

```
var v : Value = Value(i = 42)
```

Checking the active case

Use `is` to test which alternative is active:

```
if (v is i) {
    print("it's an int\n")
}
```

Use `as` to extract the value (panics if wrong):

```
let n = v as i       // 42
```

Use `?as` for safe access with a fallback:

```
let maybe_f = v ?as f ?? 0.0
```

The `variant_index()` function returns the zero-based index of the active case.

See also:

Tuples, *Variants* in the language reference.

Full source: `tutorials/language/11_tuples_and_variants.das`

Next tutorial: *Function Pointers*

5.1.12 Function Pointers

This tutorial covers getting a pointer to a named function with `@@`, the `function<>` type, calling function pointers, disambiguating overloads with `@@<signature>`, anonymous functions, and the distinction between `function<>`, `lambda<>`, and `block<>`.

Getting a function pointer

Use `@@` before a function name to get a pointer to it. The result is a value of type `function<>`:

```
def double_it(x : int) : int {
    return x * 2
}

let fn = @@double_it       // type: function<(x:int):int>
print("{fn(5)}\n")         // 10
```

Function pointers support **call syntax** — call them like regular functions. `invoke()` is the explicit alternative:

```
fn(5)              // call syntax
invoke(fn, 5)      // same result
```

Passing functions as arguments

Declare a parameter with `function<>` type to accept function pointers:

```
def apply(f : function<(x:int):int>; value : int) : int {
    return f(value)
}

apply(@@double_it, 7)        // 14
apply(@@negate, 7)           // -7
```

Only `@@`-created values can be passed to `function<>` parameters. Lambdas (`@`) and blocks (`$`) are rejected by the compiler.

Disambiguating overloads

When multiple functions share a name, specify the signature:

```
def add(a, b : int) : int { return a + b }
def add(a, b : float) : float { return a + b }

let fn_int   = @@<(a:int;b:int):int> add
let fn_float = @@<(a:float;b:float):float> add

fn_int(3, 4)        // 7
fn_float(1.5, 2.5)  // 4
```

Anonymous functions

`@@` can also create a function inline — no name, no capture:

```
let square <- @@(x : int) : int => x * x
print("{square(6)}\n")        // 36
```

Multi-line version:

```
let clamp <- @@(x : int) : int {
    if (x < 0) { return 0 }
    if (x > 100) { return 100 }
    return x
}
```

Anonymous functions **cannot** capture outer variables:

```
var outer = 10
let bad <- @@(x : int) : int => x + outer   // ERROR: can't locate variable
```

Pass anonymous functions directly to `function<>` parameters:

```
apply(@@(x : int) : int => x * x + 1, 8)    // 65
```

function<> vs lambda<> vs block<>

These are three **distinct** callable types:

Feature	function<> (@@)	lambda<> (@)	block<> ($)
Allocation	None	Heap	Stack
Capture	None	By copy	By reference
Storable	Yes	Yes (move)	No
Returnable	Yes	Yes	No

A function<> parameter accepts **only** @@ values. A lambda<> parameter accepts **only** @ values. A block<> parameter accepts any of them (most flexible).

Use function<> for pure transforms with no state. Use lambda<> when you need captured variables. Use block<> for temporary callback parameters.

See also:

Functions in the language reference.

Full source: tutorials/language/12_function_pointers.das

Next tutorial: *Blocks*

5.1.13 Blocks

This tutorial covers block declarations, passing blocks to functions, the pipe <| operator, simplified => syntax, call syntax, and invoke() for local block variables.

Declaring a block

Blocks are anonymous functions prefixed with $():

```
var blk = $(a, b : int) => a + b
print("{invoke(blk, 3, 4)}\n")    // 7
```

Blocks as function arguments

Use block in the parameter type. Inside the function, call blocks directly — just like regular functions:

```
def do_twice(action : block) {
    action()           // call syntax - preferred
    action()
}
```

With typed parameters and return:

```
def apply_to(value : int; transform : block<(x:int):int>) : int {
    return transform(value)    // call syntax, not invoke
}
```

The pipe operator

Use <| to pass a trailing block argument:

```
do_twice() {
    print("hello!\n")
}
```

Simplified blocks

When a block has a single expression body, use =>:

```
let result = apply_to(5, $(x) => x * x)   // 25
```

Call syntax vs invoke

Inside a function that receives a block parameter, use **call syntax**:

```
def filter(arr : array<int>; predicate : block<(x:int):bool>) {
    for (v in arr) {
        if (predicate(v)) {      // call syntax: predicate(v)
            print("{v} ")
        }
    }
}
```

For **local block variables**, use `invoke()`:

```
var blk = $(a, b : int) => a + b
print("{invoke(blk, 3, 4)}\n")
```

Capture rules

Blocks capture outer variables **by reference** and are stack-allocated. They cannot be stored in containers or returned from functions — use *lambdas* for that.

See also:

Functions in the language reference.

Full source: `tutorials/language/13_blocks.das`

Next tutorial: *Lambdas and Closures*

5.1.14 Lambdas and Closures

This tutorial covers lambda declarations with `@()`, capture modes (copy, move, clone), call syntax, storing lambdas, and how lambdas differ from blocks and function pointers.

Declaring a lambda

Lambdas are prefixed with `@()` and are heap-allocated:

```
var mul <- @(x : int) : int { return x * 3 }
print("{mul(10)}\n")    // 30
```

Simplified syntax with =>:

```
var add <- @(a, b : int) : int => a + b
```

Call syntax

Lambda variables support **call syntax** — call them like regular functions:

```
var doubler <- @(x : int) : int => x * 2
print("{doubler(7)}\n")    // 14
```

`invoke()` is the explicit alternative — same behavior:

```
print("{invoke(doubler, 7)}\n")    // 14
```

Prefer call syntax; use `invoke()` when you need it.

Capture modes

By default, lambdas capture outer variables **by copy**:

```
var multiplier = 10
var fn <- @(x : int) : int { return x * multiplier }
// changing multiplier here does not affect fn
```

Use `capture()` for other modes:

- `capture(move(var))` — transfers ownership; source is zeroed
- `capture(clone(var))` — deep copy; source unchanged
- `capture(ref(var))` — by reference (requires `unsafe`)

```
var data : array<int>
data |> push(1)
unsafe {
    var fn2 <- @capture(move(data)) () { print("{data}\n") }
}
```

Storing lambdas

Lambdas must be **moved** with <-, not copied:

```
var a <- @() { print("hello\n") }
var b <- a          // a is now empty
b()
```

Lambdas cannot be copied, but they **can** be stored in arrays using `emplace`, which moves the lambda into the container:

```
var callbacks : array<lambda<():void>>
var greet <- @() { print("hello from callback\n") }
callbacks |> emplace(greet)    // greet is now empty
var farewell <- @() { print("goodbye from callback\n") }
callbacks |> emplace(farewell)
for (cb in callbacks) {
    invoke(cb)
}
```

Blocks **cannot** be stored in arrays or variables — they live on the stack and are only valid as function arguments.

Stateful lambda factory

A function can create and return a lambda that captures state:

```
def make_counter() : lambda<():int> {
    var count = 0
    return <- @capture(clone(count)) () : int {
        count += 1
        return count
    }
}
```

Each call creates an independent counter.

Lambda lifecycle and finally blocks

A lambda is a heap-allocated struct. The full lifecycle is:

1. **Capture** — outer variables are copied/moved/cloned into the struct
2. **Invoke** — the lambda body runs (may be called many times)
3. **Destroy** — when the lambda is deleted or garbage collected: a) The `finally{}` block runs (user cleanup code) b) Captured fields are finalized (compiler-generated `delete`) c) The struct memory is freed

Captured fields are automatically deleted on destruction unless:

- The field was captured **by reference** (not owned)
- The field was captured **by move/clone** with `doNotDelete`
- The field type is POD (int, float — no cleanup needed)

Finally block

A lambda's `finally{}` block runs **once** when the lambda is **destroyed** — not after each invocation. This is different from a *block* `finally{}`, which runs after every call:

```
var demo <- @() {
    print("body\n")
} finally {
    print("destroyed\n")    // runs once, on deletion
}
demo()     // prints "body"
demo()     // prints "body"
unsafe { delete demo; }   // prints "destroyed"
```

Use lambda `finally{}` for one-time destruction cleanup (releasing resources, closing handles). For per-call cleanup, use scoped variables or `defer` inside the lambda body.

Lambda vs block vs function pointer

Feature	`function<>` (@@)	`lambda<>` (@)	`block<>` ($)
Allocation	None	Heap	Stack
Capture	None	By copy (default)	By reference
Storable	Yes	Yes (move only)	No
Returnable	Yes	Yes	No

A `function<>` parameter accepts only `@@` values. A `lambda<>` parameter accepts only `@` values. A `block<>` parameter accepts any of them (most flexible).

See *Function Pointers* for `@@` and `function<>` details.

See also:

Lambdas, *Functions* in the language reference.

Full source: `tutorials/language/14_lambdas.das`

Next tutorial: *Iterators and Generators*

5.1.15 Iterators and Generators

This tutorial covers built-in iterators (`range`, `each`, `keys`, `values`), creating generators with `yield`, and making custom types iterable.

Built-in iterators

Arrays, ranges, tables, and strings are all iterable:

```
for (i in range(5)) { ... }
for (v in my_array) { ... }
for (k, v in keys(table), values(table)) { ... }
```

Generators

A generator lazily produces values one at a time using `yield`. Generators use `$` (either `$ [ ]` or `$() [ ]`):

```
var counter <- generator<int>() <| $ {
    for (i in range(5)) {
        yield i
    }
    return false
}

for (v in counter) {
    print("{v} ")       // 0 1 2 3 4
}
```

Generators maintain state between yields — local variables keep their values across calls.

Generator patterns

Filtering — yield only matching values:

```
var evens <- generator<int>() <| $ {
    for (i in range(10)) {
        if (i % 2 == 0) { yield i }
    }
    return false
}
```

Stateful — maintain running computations:

```
var fibs <- generator<int>() <| $ {
    var a = 0
    var b = 1
    while (a < 100) {
        yield a
        let next = a + b
        a = b
        b = next
    }
    return false
}
```

Custom iterable types

Define an `each()` function to make any struct iterable:

```
struct NumberRange {
    low : int
    high : int
}

def each(r : NumberRange) : iterator<int> {
```

(continues on next page)

(continued from previous page)

```
    return <- generator<int>() <| $ {
        for (i in r.low .. r.high) {
            yield i
        }
        return false
    }
}

var r = NumberRange(low=3, high=8)
for (i in r) {
    print("{i} ")       // 3 4 5 6 7
}
```

Note: Generators are **one-shot** — once exhausted they produce no more values. Create a new generator to iterate again.

See also:

Iterators, *Generators* in the language reference.

Full source: `tutorials/language/15_iterators_and_generators.das`

Next tutorial: *Modules and Program Structure*

5.1.16 Modules and Program Structure

This tutorial covers the file layout (options, module, require), creating and using a separate module file, public vs private visibility, qualified calls with `::`, and common standard library modules.

File layout

A typical daslang file follows this order:

```
options gen2                          // compiler options
module my_module public               // module declaration
require math                          // imports
// ... declarations (struct, enum, def, etc.)
```

The `module` line must appear before any declarations.

Module declaration

The `module` keyword names the current file. Omit it to default to the filename:

```
module my_lib public      // public declarations by default
module my_lib private     // private declarations by default
module my_lib shared      // shared across contexts
```

Creating a module

Create a file `tutorial_helpers.das` next to your main script:

```
options gen2
module tutorial_helpers public

struct Item {
    name : string
    value : int
}

def make_item(n : string; v : int) : Item {
    return Item(name=n, value=v)
}

def describe(item : Item) : string {
    return "{item.name} (worth {item.value})"
}

def private internal_id(item : Item) : int {
    return item.value * 31
}

let MAX_ITEMS = 100
```

Using a module

In the main file, import and use it:

```
require tutorial_helpers

let sword = make_item("Sword", 150)
print("{describe(sword)}\n")
print("max: {MAX_ITEMS}\n")
```

If the module file is in the same directory, `require` finds it by name.

Qualified calls

Use `module::function` to disambiguate:

```
let shield = tutorial_helpers::make_item("Shield", 80)
let r = math::sqrt(16.0)
```

This is essential when two modules export the same function name.

Visibility

Each declaration can override the module default:

```
def public  helper() { ... }    // always visible
def private internal() { ... }  // only within this module
```

Private functions like `internal_id()` above cannot be called from other modules — the compiler reports an error.

Re-exporting

`require ... public` re-exports a module to your importers:

```
require math public       // importers of this module also get math
```

Common standard library modules

Module	Purpose
`math`	sin, cos, sqrt, atan2, etc.
`strings`	to_int, to_float, starts_with, etc.
`fio`	File I/O
`rtti`	Runtime type information
`daslib/json`	JSON parsing
`daslib/regex`	Regular expressions
`daslib/algorithm`	Sorting and searching
`daslib/strings_boost`	Advanced string utilities

See also:

Modules, *Program Structure* in the language reference.

Full source: `tutorials/language/16_modules.das` and `tutorials/language/tutorial_helpers.das`

Next tutorial: *Move, Copy, and Clone*

5.1.17 Move, Copy, and Clone

This tutorial covers the three assignment operators: copy (=), move (<-), and clone (:=), when to use each, and how types determine operator availability.

Copy (=)

Bitwise copy. Source unchanged. Works for POD types (int, float, string) and POD-only structs:

```
var a = 42
var b = a         // b is 42, a is still 42
```

Move (<-)

Transfers ownership. Source is zeroed. Use for containers (array, table), lambdas, and iterators:

```
var nums <- [1, 2, 3]
var other <- nums       // other owns the array, nums is empty
```

Functions returning containers must use `return <-`:

```
def make_data() : array<int> {
    var result : array<int>
    result |> push(1)
    return <- result
}
```

Clone (:=)

Deep copy. Both sides remain valid and independent:

```
var original : array<int>
original |> push(1)
original |> push(2)

var copy : array<int>
copy := original        // deep copy
copy |> push(999)       // does not affect original
```

Clone initialization:

```
var another := original
```

Type compatibility

Type	=	<-	:=
int, float, POD	Yes	Yes	Yes
string	Yes	Yes	Yes
array, table	No	Yes	Yes
lambda	No	Yes	No
iterator	No	Yes	No
block	No	No	No

Relaxed assign

By default, the compiler auto-promotes = to <- when the right side is temporary (literal, function return). Disable with `options relaxed_assign = false`.

Struct field initialization

Each field can use a different mode:

```
var weapons : array<string>
weapons |> push("bow")

var loadout = Inventory(items <- weapons, weight = 5)
// weapons is now empty after the move
```

Custom clone functions

Override clone behavior by defining a `clone` function for your type:

```
struct GameState {
    level : int
    score : int
}

def clone(var dst : GameState; src : GameState) {
    dst.level = src.level + 1000   // custom logic during clone
    dst.score = src.score
}

var original = GameState(level=5, score=100)
var copy := original    // invokes custom clone
// copy.level is 1005, not 5
```

See also:

Move, Copy, and Clone in the language reference.

Full source: `tutorials/language/17_move_copy_clone.das`

Next tutorial: *Classes and Inheritance*

5.1.18 Classes and Inheritance

This tutorial covers defining classes, constructors, `super()` and `super.method()`, abstract and virtual methods, inheritance with `override`, polymorphism, private members, and static fields/methods.

Defining a class

Classes are like structs with virtual methods and inheritance:

```
class Shape {
    name : string

    def Shape(n : string) {
        name = n
    }

    def abstract area : float

    def describe {
        print("{name}: area = {area()}\n")
    }
}
```

Inheritance and super

Single-parent inheritance with `class Derived : Base`. Use `super()` to call the parent constructor:

```
class Circle : Shape {
    radius : float

    def Circle(r : float) {
        super("Circle")       // calls Shape`Shape(self, "Circle")
        radius = r
    }

    def override area : float {
        return 3.14159 * radius * radius
    }
}
```

Use `super.method()` to call the parent's version of a method, bypassing virtual dispatch:

```
class Rectangle : Shape {
    width : float
    height : float

    def Rectangle(w, h : float) {
        super("Rectangle")
        width = w
        height = h
    }

    def override describe {
        super.describe()      // calls Shape`describe(self)
        print("  (w={width}, h={height})\n")
    }
}
```

The compiler rewrites `super()` to `Parent`Constructor(self, ...)` and `super.method()` to `Parent`method(self, ...)`.

Creating instances

Use `new` to create a class on the heap:

```
var c = new Circle(5.0)
c.describe()
```

Polymorphism

Base pointers hold any derived instance. Method calls dispatch virtually:

```
var shapes : array<Shape?>
shapes |> push(new Circle(3.0))
shapes |> push(new Rectangle(2.0, 5.0))

for (s in shapes) {
    s.describe()    // calls the correct area() for each type
}
```

Private members

```
class Counter {
    private count : int = 0
    def increment { count += 1 }
    def get_count : int { return count }
}
```

Static fields and methods

`static` fields are shared across all instances. `def static` methods access static fields but not `self`:

```
class Tracker {
    static total : int = 0
    id : int

    def Tracker {
        total += 1
        id = total
    }

    def static getTotal : int {
        return total
    }

    def static reset {
        total = 0
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Call static methods with backtick syntax:

```
print("{Tracker`getTotal()}\n")
Tracker`reset()
```

Key concepts

- `abstract` — subclasses **must** implement
- `override` — replaces a parent method
- `sealed` — prevents further overriding or inheritance
- `self` — implicit pointer to current instance
- `super()` — calls parent constructor
- `super.method()` — calls parent's version of a method
- `def static` — method without self, accesses static fields only
- `ClassName`method()` — call static methods from outside

Note: Structs can also have methods via the `[class_method]` annotation from `daslib/class_boost`, which adds an implicit `self` parameter to static functions on structs.

See also:

Classes, *Structs* in the language reference.

Full source: `tutorials/language/18_classes.das`

Next tutorial: *Generic Programming*

5.1.19 Generic Programming

This tutorial covers generic functions (omitting types / using `auto`), named type variables with `auto(T)`, using `auto(T)` for local variables, specialization from generic to specific, compile-time type inspection with `typeinfo`, and conditional compilation with `static_if`.

Generic functions

Omit a type to make a function generic. It is instantiated for each concrete type at the call site:

```
def print_value(v) {
    print("value = {v}\n")
}

print_value(42)       // instantiated for int
print_value(3.14)     // instantiated for float
print_value("hello")  // instantiated for string
```

Named auto

Use `auto(T)` to name a type variable. Multiple parameters sharing the same name must have the same type:

```
def add(a, b : auto(T)) : T {
    return a + b
}

add(10, 20)       // ok: both int
add(1.5, 2.5)     // ok: both float
// add(1, 2.0)    // error: int != float
```

auto(T) for local variables

Use the named type `T` to declare local variables of the same type:

```
def make_pair(a, b : auto(T)) {
    var first : T = a
    var second : T = b
    print("pair: {first}, {second}\n")
}

make_pair(10, 20)           // T = int
make_pair("hello", "world") // T = string
```

default<T>

Use `default<T>` to create a default-initialized value of any type. This is useful in generic code where you don't know the concrete type:

```
def sum_array(arr : array<auto(T)>) : T {
    var total = default<T>    // 0 for int, 0.0 for float, "" for string
    for (v in arr) {
        total += v
    }
    return total
}
```

Specialization

When multiple overloads match, the compiler picks the **most specific** one. This lets you write a generic fallback and optimize for known types:

```
// Most generic: takes anything
def process(v) {
    print("anything: {v}\n")
}

// More specific: any array (T is the element type)
def process(arr : array<auto(T)>) {
```

(continues on next page)

(continued from previous page)

```
    print("array of {typeinfo typename(type<T>)}\n")
}

// Most specific: array<int> (exact match)
def process(arr : array<int>) {
    var total = 0
    for (v in arr) { total += v }
    print("array<int>, sum={total}\n")
}
```

Calling these:

```
process(42)           // → anything
process(["a", "b"])   // → array of string
process([1, 2, 3])    // → array<int>
```

typeinfo

Query types at compile time:

```
def describe_type(v : auto(T)) {
    let name = typeinfo typename(type<T>)
    let size = typeinfo sizeof(type<T>)
    print("type={name}, size={size}\n")
}
```

Common typeinfo traits:

- `typeinfo typename(expr)` — human-readable type name
- `typeinfo sizeof(expr)` — size in bytes
- `typeinfo is_numeric(expr)` — true for int, float, etc.
- `typeinfo is_string(expr)` — true for string
- `typeinfo is_array(expr)` — true for `array<T>`
- `typeinfo has_field<name>(expr)` — true if struct has field

static_if

Select code paths based on type traits — resolved at compile time:

```
def to_string_generic(v : auto(T)) : string {
    static_if (typeinfo is_numeric(type<T>)) {
        return "number: {v}"
    } else {
        return "other: {v}"
    }
}
```

Struct introspection

Use `has_field` to write functions that work on multiple struct types:

```
def get_name(obj) : string {
    static_if (typeinfo has_field<name>(obj)) {
        return obj.name
    } else {
        return "unnamed"
    }
}
```

See also:

Generic Programming in the language reference.

Full source: `tutorials/language/19_generics.das`

Next tutorial: *Lifetime and Cleanup*

5.1.20 Lifetime and Cleanup

This tutorial covers daslang's memory model, explicit cleanup with `delete`, automatic cleanup with `var inscope`, custom finalizers, and `finally` blocks.

Memory model

daslang does **not** automatically clean up local variables when they go out of scope. There is no implicit destructor call. Containers (arrays, tables) will have their memory reclaimed eventually, but finalizers are not called automatically.

For deterministic cleanup (closing files, releasing locks), use:

- `delete` — explicit immediate cleanup
- `var inscope` — automatic cleanup at end of scope

For simple programs, you often don't need either — the process reclaims all memory on exit. But for long-running applications or resource management, use the tools below.

Explicit delete

`delete` zeroes a variable and calls its finalizer. For containers it frees all elements:

```
var data <- [1, 2, 3]
delete data           // data is now empty
```

Custom finalizers

Define a `finalize` function for custom cleanup logic:

```
struct Resource {
    name : string
    id : int
}

def finalize(var r : Resource) {
    print("cleanup: {r.name}\n")
}

var res = Resource(name="DB", id=1)
delete res            // prints "cleanup: DB"
```

var inscope

`var inscope` automatically calls `delete` at the end of the enclosing block — like RAII in C++ or `defer` in Go. This is the recommended approach for deterministic cleanup:

```
if (true) {
    var inscope r1 = acquire("File", 2)
    var inscope r2 = acquire("Socket", 3)
    // ... use resources ...
    // delete r2, then delete r1, called automatically (reverse order)
}
```

Use `inscope` for file handles, connections, locks, or any resource that needs prompt release.

finally blocks

`finally` runs cleanup code when a block exits:

```
for (i in range(3)) {
    counter += 1
} finally {
    print("done, counter={counter}\n")
}
```

Heap pointers

For class instances created with `new`, `delete` requires `unsafe`:

```
var p = new MyClass()
// ... use p ...
unsafe { delete p }
```

Or use `var inscope` for automatic cleanup:

```
unsafe { // needs 'unsafe', because deleting classes is unsafe
  var inscope p = new MyClass()
}
// deleted automatically at end of scope
```

When to use what

Approach	When to use
Do nothing	Short-lived programs where process exit reclaims memory
`var inscope`	Deterministic cleanup (files, sockets, locks)
`delete`	Immediate memory reclaim in long-running code
Custom `finalize`	Cleanup logic (logging, counters, etc.)

Note: For heap pointers (from `new`), `delete` requires `unsafe`. For local variables, `delete` is safe.

See also:

Finalizers, *Statements* in the language reference.

Full source: `tutorials/language/20_lifetime.das`

Next tutorial: *Error Handling*

5.1.21 Error Handling

This tutorial covers daslang's error-handling mechanisms: `panic` for fatal errors, `try/recover` for catching panics, assertions for invariants, and `defer` for cleanup actions.

panic

`panic("message")` halts execution immediately with a runtime error. Use it for unrecoverable situations:

```
panic("something went horribly wrong")
```

Certain operations (null dereference, out-of-bounds access) cause implicit panics.

try / recover

`try/recover` catches panics and continues execution. It is **not** general exception handling — only panics are caught:

```
try {
    risky_operation()
} recover {
    print("recovered from panic\n")
}
```

Note: You cannot `return` from inside `try` or `recover` blocks. Assign results to a variable instead.

assert and verify

`assert(condition, "message")` checks a condition at runtime. It may be stripped in release builds:

```
assert(x > 0, "x must be positive")
```

`verify(condition, "message")` is never stripped — side effects in the condition are always evaluated:

```
verify(initialize(), "init failed")
```

static_assert and concept_assert

`static_assert` checks conditions at compile time:

```
static_assert(typeinfo(sizeof type<int>) == 4, "expected 4-byte int")
```

`concept_assert` works like `static_assert` but reports the error at the **caller's** line — useful inside generic functions:

```
def sum(arr : auto(TT)[]) {
    concept_assert(typeinfo is_numeric(type<TT>), "need numeric type")
}
```

finally and defer

`finally` runs cleanup code when a block exits:

```
for (i in range(3)) {
    total += i
} finally {
    print("loop done\n")
}
```

`defer` (from `daslib/defer`) schedules cleanup that runs when the enclosing scope exits. Multiple defers run in LIFO order:

```
require daslib/defer

def example {
    defer() { print("third\n") }
    defer() { print("second\n") }
    defer() { print("first\n") }
}
// prints: first, second, third (reverse order)
```

See also:

Statements, *Unsafe* in the language reference.

Full source: `tutorials/language/21_error_handling.das`

Next tutorial: *Unsafe and Pointers*

5.1.22 Unsafe and Pointers

This tutorial covers daslang's `unsafe` block for opt-in low-level operations: raw pointers, `addr`/`deref`, null handling, pointer arithmetic, and `reinterpret` casts.

unsafe blocks

`unsafe { ... }` enables pointer operations and other low-level features that are normally disallowed:

```
var x = 42
unsafe {
    var p : int? = addr(x)
    *p = 100
}
```

Use `unsafe(expr)` for a single expression:

```
let p = unsafe(addr(x))
```

addr and deref

`addr(x)` returns a pointer `T?` to a local variable. `*p` (or `deref(p)`) follows the pointer:

```
unsafe {
    var p = addr(x)
    print("{*p}\n")   // prints value of x
    *p = 999          // modifies x through pointer
}
```

Null pointers

Dereferencing a null pointer causes a panic inside `try`/`recover`:

```
var np : int? = null
try {
    print("{*np}\n")
} recover {
    print("caught null deref\n")
}
```

Pointer arithmetic

Pointers can index into contiguous memory (e.g., arrays):

```
unsafe {
    var arr = [10, 20, 30]
    var p = addr(arr[0])
    print("{p[0]}, {p[1]}, {p[2]}\n")  // 10, 20, 30
}
```

reinterpret

`reinterpret` performs a raw bit cast between types of the same size:

```
let f = 1.0
let bits = unsafe(reinterpret<int> f)
print("{bits:x}\n")  // 3f800000
```

> **Warning:** `unsafe` does not propagate into lambdas or generators. Each lambda/generator body needs its own `unsafe` block if needed.

See also:

Unsafe in the language reference.

Full source: `tutorials/language/22_unsafe.das`

Next tutorial: *String Builder and Formatting*

5.1.23 String Builder and Formatting

This tutorial covers format specifiers in string interpolation and programmatic string construction with `build_string`.

Format specifiers

Inside `{expr}` interpolation, add `:flags width.precision type`:

```
let x = 42
print("{x:08x}\n")      // 0000002a
print("{3.14:.2f}\n")   // 3.14
```

Flags:

- `0` — zero-pad
- `-` — left-align
- `+` — force sign

Type characters:

- `d/i` — signed decimal, `u` — unsigned decimal
- `x/X` — hex lower/upper, `o` — octal
- `f` — fixed-point float, `e/E` — scientific
- `g/G` — general (shortest of `f` or `e`)

Escaping braces

Use \{ and \} for literal curly braces:

```
print("\{value\}\n")      // prints: {value}
print("\{{x}\}\n")        // prints: {42}
```

build_string

`build_string` constructs a string efficiently using a writer:

```
require strings

var csv = build_string() $(var writer : StringBuilderWriter) {
    writer |> write("name,score\n")
    writer |> write("Alice,95\n")
}
```

Writer functions:

- `write(value)` — write any printable value
- `write_char(ch)` — write a single character
- `write_chars(ch, count)` — write a character N times
- `write_escape_string(str)` — write with escape sequences visible

See also:

String builder in the language reference.

Full source: `tutorials/language/23_string_format.das`

Next tutorial: *Pattern Matching*

5.1.24 Pattern Matching

This tutorial covers daslang's pattern matching with the `match` macro from `daslib/match`.

Basic match

`match(expr) { ... }` dispatches on values. Each arm is an `if`:

```
require daslib/match

def describe(n : int) : string {
    match (n) {
        if (0) { return "zero" }
        if (1) { return "one" }
        if (_) { return "other" }
    }
    return "unreachable"
}
```

Note: `match` works on functions that return values.

Wildcards and binding

- `_` — wildcard, matches anything
- `$v(name)` — bind matched value to a variable:

```
match (point) {
    if (Point(x = 0, y = 0)) { return "origin" }
    if (Point(x = $v(x), y = $v(y))) { return "({x}, {y})" }
}
```

Guards

Add && after a pattern for extra conditions:

```
match (shape) {
    if (Shape(size = $v(sz)) && sz > 100) {
        return "large"
    }
}
```

OR patterns

Match multiple alternatives with `||`:

```
match (color) {
    if (Color.red || Color.green || Color.blue) {
        return true
    }
}
```

Variant matching

Use `$v(name) as alternative` to match variant alternatives:

```
match (value) {
    if ($v(i) as i) { return "int" }
    if ($v(f) as f) { return "float" }
}
```

Tuple matching

Match tuple elements positionally:

```
match (pair) {
    if ((0, "zero")) { return "exact" }
    if ((1, _)) { return "starts with one" }
    if (($v(n), "hello")) { return "hello #{n}" }
}
```

Array matching

Static arrays match element-by-element. Dynamic arrays match head elements; use `...` to ignore the tail:

```
match (sa) {
    if (fixed_array<int>(0, 0, 0)) { return "zeros" }
    if (fixed_array<int>(1, $v(b), $v(c))) { return "1,{b},{c}" }
}

match (da) {
    if (array<int>(0, 0, ...)) { return "starts with 0,0" }
    if (array<int>($v(x), $v(y), ...)) { return "starts with {x},{y}" }
}
```

match_expr — computed patterns

`match_expr(expression)` evaluates at runtime instead of matching literally. Useful when the expected value depends on a captured variable:

```
match (t) {
    if (($v(a), match_expr(a + 1), match_expr(a + 2))) {
        return true   // matches consecutive triples
    }
}
```

Bool matching

`match` works on booleans:

```
match (b) {
    if (true) { return "yes" }
    if (false) { return "no" }
}
```

multi_match and static_match

- `multi_match` — all matching arms execute (not just first)
- `static_match` — silently drops type-mismatched arms (for generics)

```
multi_match (n) {
    if ($v(a) && a > 0) { tags += " positive" }
    if ($v(a) && (a % 2 == 0)) { tags += " even" }
}
```

See also:

Pattern matching in the language reference.

Full source: `tutorials/language/24_pattern_matching.das`

Next tutorial: *Annotations and Options*

5.1.25 Annotations and Options

This tutorial covers function and struct annotations, and compiler options that control lint, safety, and optimization.

Function annotations

Annotations are placed in `[ ]` before the function definition:

```
[export]
def main { ... }

[sideeffects]
def log(msg : string) { print(msg) }

[unused_argument(y)]
def use_only_x(x, y : int) : int { return x * 2 }
```

Common function annotations:

- `[export]` — callable from host application
- `[sideeffects]` — has side effects, won't be eliminated
- `[init]` — runs at context initialization
- `[finalize]` — runs at context shutdown
- `[deprecated(message="...")]` — produces warning on use
- `[unused_argument(x)]` — suppress unused arg lint
- `[unsafe_operation]` — calling requires `unsafe` block
- `[no_aot]` — disable AOT for this function
- `[jit]` — request JIT compilation

Combining annotations

Separate multiple annotations with commas:

```
[export, sideeffects]
def annotated_func() { ... }
```

Struct annotations

- `[safe_when_uninitialized]` — zero-filled memory is valid
- `[cpp_layout]` — C++ struct alignment
- `[persistent]` — survives context reset

Private functions

Use the `private` keyword:

```
def private helper() : string {
    return "module-private"
}
```

Compiler options

Set at the top of a file:

```
options gen2
options no_unused_function_arguments
options stack = 32768
```

Key options:

Option	Description
`gen2`	Enable gen2 syntax
`no_unused_function_arguments`	Unused args become errors
`strict_smart_pointers`	Strict smart pointer checks (default: on)
`no_global_variables`	Disallow module-level globals
`unsafe_table_lookup`	`tab[key]` requires unsafe (default: on)
`stack = N`	Stack size in bytes
`no_aot`	Disable AOT for module
`lint`	Enable/disable all lint checks
`no_deprecated`	Deprecated usage becomes error

See also:

Annotations, *Options* in the language reference.

Full source: `tutorials/language/25_annotations_and_options.das`

Next tutorial: *Contracts*

5.1.26 Contracts

This tutorial covers contract annotations from `daslib/contracts` for constraining generic function arguments at compile time.

What are contracts?

Contracts are compile-time constraints on generic function parameters. When a generic function is called, the contract checks whether each argument satisfies the type constraint. If not, the function is skipped during overload resolution.

Basic usage

```
require daslib/contracts

[expect_any_numeric(a), expect_any_numeric(b)]
def safe_add(a, b) {
    return a + b
}
// safe_add(3, 4)       - works
// safe_add("a", "b")   - no match, not numeric
```

Built-in contracts

Always available (no import needed):

- `[expect_ref(arg)]` — must be a reference
- `[expect_dim(arg)]` — must be a fixed-size array
- `[expect_any_vector(arg)]` — must be a C++ vector type

Library contracts

From `daslib/contracts`:

- `[expect_any_array(a)]` — `array<T>` or `T[N]`
- `[expect_any_numeric(a)]` — `int`, `float`, etc.
- `[expect_any_enum(a)]` — any enumeration
- `[expect_any_struct(a)]` — any struct (not class)
- `[expect_any_tuple(a)]` — any tuple
- `[expect_any_variant(a)]` — any variant
- `[expect_any_function(a)]` — any function type
- `[expect_any_lambda(a)]` — any lambda
- `[expect_any_bitfield(a)]` — any bitfield
- `[expect_pointer(a)]` — any pointer
- `[expect_class(a)]` — class pointer

Combining contracts

Use logical operators inside the brackets:

```
[expect_any_numeric(a) && expect_any_numeric(b)]   // AND
[expect_any_tuple(x) || expect_any_variant(x)]     // OR
[!expect_any_numeric(x)]                            // NOT
[expect_any_tuple(a) ^^ expect_any_variant(b)]     // XOR
```

concept_assert

`concept_assert(condition, "message")` is a compile-time check that reports errors at the **caller's** line (not inside the generic):

```
def sum_all(arr : array<auto(TT)>) : TT {
    concept_assert(typeinfo is_numeric(type<TT>),
        "sum_all requires numeric elements")
    // ...
}
```

See also:

Generic programming in the language reference.

Full source: `tutorials/language/26_contracts.das`

Next tutorial: *Testing with dastest*

5.1.27 Testing with dastest

This tutorial covers the daslang testing framework: writing tests with `[test]`, assertion functions, sub-tests, and running tests.

Setup

Import the testing framework:

```
require dastest/testing_boost public
```

Writing tests

Annotate functions with `[test]`. Each test receives a `T?` context:

```
[test]
def test_arithmetic(t : T?) {
    t |> equal(2 + 2, 4)
    t |> equal(3 * 3, 9, "multiplication")
}
```

Assertions

Function	Behavior
`t |> equal(a, b)`	Check `a == b`; non-fatal on failure
`t |> strictEqual(a, b)`	Check `a == b`; **fatal** on failure (stops test)
`t |> success(cond)`	Check condition is truthy; non-fatal
`t |> failure("msg")`	Always fails; non-fatal
`t |> accept(value)`	Suppress unused warnings (no assertion)

All assertion functions accept an optional message string as the last argument.

Sub-tests

Group related checks with `run`:

```
[test]
def test_strings(t : T?) {
    t |> run("concat") @@(t : T?) {
        t |> equal("ab" + "cd", "abcd")
    }
    t |> run("length") @@(t : T?) {
        t |> equal(length("hello"), 5)
    }
}
```

Use `@@(t : T?) { ... }` (no-capture lambda) for the callback.

Running tests

From the command line:

```
daslang.exe dastest/dastest.das -- --test path/to/test.das
```

To test an entire directory:

```
daslang.exe dastest/dastest.das -- --test path/to/tests/
```

See also:

Full source: `tutorials/language/27_testing.das`

Next tutorial: *LINQ — Language-Integrated Query*

5.1.28 LINQ — Language-Integrated Query

This tutorial covers daslang's LINQ module for C#-style query operations on iterators and arrays, plus the `linq_boost` macros for composing pipelines.

Setup

Import both modules:

```
require daslib/linq
require daslib/linq_boost
```

Most LINQ functions come in several flavors:

- `func(iterator, ...)` — returns a lazy iterator
- `func(array, ...)` — returns a new array
- `func_to_array(iterator, ...)` — materializes into an array
- `func_inplace(var array, ...)` — mutates in place

Filtering

`where_` filters elements by a predicate (the trailing underscore avoids collision with the built-in keyword):

```
var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
var evens = where_to_array(numbers.to_sequence(), $(x) => x % 2 == 0)
// evens: [4, 2, 6]
```

The `_where` shorthand uses `_` as the element placeholder:

```
var big = _where_to_array(numbers.to_sequence(), _ > 4)
// big: [5, 9, 6, 5]
```

Projection

`select` transforms each element:

```
var doubled = _select_to_array(numbers.to_sequence(), _ * 2)
// doubled: [6, 2, 8, 2, 10, 18, 4, 12, 10]
```

`select_many` flattens nested sequences into one:

```
var nested = [
    ["a", "b", "c"].to_sequence(),
    ["d", "e", "f"].to_sequence()
]
var flat = select_many_to_array(
    nested.to_sequence(),
    $(s : string) => "{s}!"
)
// flat: ["a!", "b!", "c!", "d!", "e!", "f!"]
```

Sorting

`order` sorts ascending, `order_descending` sorts descending:

```
var unsorted = [5, 3, 8, 1, 4]
var asc = order(unsorted)            // [1, 3, 4, 5, 8]
var desc = order_descending(unsorted) // [8, 5, 4, 3, 1]
```

Sort structs by a field with `_order_by`:

```
var by_age = _order_by(team.to_sequence(), _.age)
```

Partitioning

- `skip(arr, n)` / `take(arr, n)` — from the front
- `skip_last(arr, n)` / `take_last(arr, n)` — from the end
- `skip_while` / `take_while` — by predicate
- `chunk(iter, n)` — split into groups of N

```
var seq = [for (x in 0..8); x]
skip(seq, 3)        // [3, 4, 5, 6, 7]
take(seq, 3)        // [0, 1, 2]
take_last(seq, 3)   // [5, 6, 7]
skip_last(seq, 3)   // [0, 1, 2, 3, 4]
```

Aggregation

Function	Description
`count(arr)`	Number of elements
`count(arr, pred)`	Number of elements matching predicate
`sum(iter)`	Sum of elements
`average(iter)`	Arithmetic mean
`min(iter)` / `max(iter)`	Minimum / maximum
`aggregate(iter, seed, func)`	Custom fold with seed

```
var vals = [10, 20, 30, 40, 50]
count(vals)                          // 5
count(vals, $(v) => v > 25)          // 3
sum(vals.to_sequence())              // 150
aggregate(vals.to_sequence(), 1, $(acc, v) => acc * v)  // 12000000
```

The `_count` shorthand works like `_where`:

```
_count(vals.to_sequence(), _ >= 30)  // 3
```

Element access

- `first` / `last` — first or last element (panics if empty)
- `first_or_default` / `last_or_default` — returns default if empty
- `element_at(iter, i)` / `element_at_or_default(iter, i)`
- `single` — returns the only element (panics if count $\neq 1$)

Querying

```
any(vals.to_sequence(), $(v) => v > 40)   // true
all(vals.to_sequence(), $(v) => v > 5)    // true
contains(vals.to_sequence(), 30)           // true
```

Set operations

```
var a = [1, 2, 2, 3, 3, 3]
var b = [2, 3, 4, 5]
distinct(a)     // [1, 2, 3]
union(a, b)     // [1, 2, 3, 4, 5]
except(a, b)    // [1]
intersect(a, b) // [2, 3]
```

Zip

`zip` merges two or three sequences into tuples:

```
var names = ["Alice", "Bob", "Charlie"]
var ages  = [25, 35, 30]
var zipped = zip(names, ages)
// zipped: [(Alice,25), (Bob,35), (Charlie,30)]

var scores = [95, 87, 91]
var zipped3 = zip(names, ages, scores)
// zipped3: [(Alice,25,95), (Bob,35,87), (Charlie,30,91)]
```

The `_fold` pipeline macro

`_fold` composes multiple LINQ operations into an optimized pipeline using dot-chaining:

```
// Sum of squares of even numbers
var result = (
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    ._where(_ % 2 == 0)
    ._select(_ * _)
    .sum()
    ._fold()
)
```

(continues on next page)

(continued from previous page)

```
// result: 220

// Top 3 unique values, descending
var top3 = (
    [5, 3, 8, 1, 4, 8, 3, 9, 2, 9]
    .order_descending()
    .distinct()
    .take(3)
    ._fold()
)
// top3: [9, 8, 5]
```

The `_fold` macro rewrites the chain into efficient imperative code at compile time, avoiding intermediate allocations.

See also:

Full source: `tutorials/language/28_linq.das`

Next tutorial: *Functional Programming*

5.1.29 Functional Programming

This tutorial covers `daslib/functional` — lazy iterator adapters and higher-order combinators for daslang.

Unlike `daslib/linq` which uses blocks and focuses on query syntax, `functional` accepts lambdas and functions, emphasising composability.

filter and map

`filter` keeps elements matching a predicate. `map` transforms each element:

```
var src <- [iterator for (x in range(8)); x]
var evens <- filter(src, @(x : int) : bool { return x % 2 == 0; })

var nums <- [iterator for (x in range(5)); x]
var squared <- map(nums, @(x : int) : int { return x * x; })
```

Both return lazy iterators — elements are produced on demand. They accept lambdas (`@`) or named functions (`@@is_even`).

Note: `filter`, `map`, `scan`, and `tap` return generators and only accept lambdas or functions — **not blocks** — because blocks cannot be captured into generators.

---

reduce and fold

`reduce` combines elements pairwise. `fold` adds an initial seed:

```
var total = reduce(src) $(a, b : int) : int { return a + b }

var product = fold(src, 1) $(acc, x : int) : int { return acc * x }
```

`reduce_or_default` returns a fallback on empty input instead of panicking:

```
var safe = reduce_or_default(src, @(a, b : int) : int { return a + b; }, -1)
```

scan — running fold

`scan` yields every intermediate accumulator value:

```
var running <- scan(src, 0, @(acc, x : int) : int { return acc + x; })
// seed=0, then 0+1=1, 1+3=4, ...
```

Iterator combinators

- `chain(a, b)` — concatenate two iterators
- `enumerate(src)` — `(index, element)` tuples
- `pairwise(src)` — consecutive pairs: `(a,b), (b,c), ...`
- `islice(src, start, stop)` — slice `[start, stop)`
- `flatten(src)` — flatten nested iterators one level

```
var en <- enumerate(names)
for (v in en) { print("{v._0}: {v._1}") }

var ca <- chain(first_half, second_half)
```

Generators

- `iterate(seed, fn)` — `seed, f(seed), f(f(seed)), ...` infinitely
- `repeat(val, n)` — repeat `val` `n` times (or forever if `n < 0`)
- `cycle(src)` — endlessly repeat an iterator

```
var powers <- iterate(1, @(x : int) : int { return x * 2; })
// 1, 2, 4, 8, 16, ...

var rep <- repeat(42, 3)   // 42, 42, 42
```

Aggregation

- `sum(src)` — sum all elements
- `any(src)` — true if any element is truthy
- `all(src)` — true if all elements are truthy

Search and split

- `find(src, fn, default)` — first match or default value
- `find_index(src, fn)` — index of first match, or `-1`
- `partition(src, fn)` — split into `(matching, non_matching)` arrays

```
var first_even = find(src, @@is_even, -1)
var parts = partition(src, @@is_even)  // parts._0 = evens, parts._1 = odds
```

Side-effects and debugging

- `for_each(src, fn/block)` — apply side-effect to every element
- `tap(src, fn)` — passthrough with side-effect (debug tool)
- `echo(x)` — print `x` and return it unchanged

```
for_each(src) $(x : int) { total += x }

var tapped <- tap(src, @(x : int) { print("debug: {x}\n"); })
```

Composition

Chain these together to build data pipelines:

```
var big <- filter(src, @(x : int) : bool { return x >= 5; })
var doubled <- map(big, @(x : int) : int { return x * 2; })
var result = fold(doubled, 0) $(acc, x : int) : int { return acc + x }
```

See also:

Full source: `tutorials/language/29_functional.das`

Next tutorial: *JSON*

LINQ tutorial for block-based query syntax.

Iterators and generators in the language reference.

Lambdas in the language reference.

5.1.30 JSON

This tutorial covers `daslib/json` and `daslib/json_boost` — parsing, building, writing, and querying JSON data in daslang.

`json` provides the core parser, writer, and `JsValue` variant type. `json_boost` adds safe access operators, struct serialization, and the `%json~` reader macro.

Parsing JSON

`read_json` takes a string and returns `JsonValue?`. If parsing fails, the error string is set and null is returned:

```
var error : string
var js = read_json("{ \"name\": \"Alice\", \"age\": 30 }", error)
if (js != null) {
    print("parsed OK\n")
}
```

The `JsonValue` struct wraps a `JsValue` variant with these cases:

- `_object` — `table<string; JsonValue?>`
- `_array` — `array<JsonValue?>`
- `_string` — `string`
- `_number` — `double` (floating point)
- `_longint` — `int64` (integer)
- `_bool` — `bool`
- `_null` — `void?`

Building JSON with JV

`JV()` constructors wrap native values into `JsonValue?`:

```
var str_val  = JV("hello")   // _string
var int_val  = JV(42)        // _longint
var flt_val  = JV(3.14)      // _number
var bool_val = JV(true)      // _bool
var null_val = JVNull()      // _null
```

Multi-value `JV` creates an array:

```
var arr = JV(1, "two", true)   // [1, "two", true]
```

Writing JSON

`write_json` converts a `JsonValue?` back to a string:

```
var js = JV(42)
print(write_json(js))   // 42

var empty : JsonValue?
print(write_json(empty))  // null
```

Safe access operators

`json_boost` provides `?.`, `?[]`, and `??` for convenient access. These never crash — they return null or a default value on missing keys:

```
var js = read_json("{ \"user\": { \"name\": \"Bob\" } }", error)

let name = js?.user?.name ?? "unknown"     // "Bob"
let age  = js?.user?.age ?? 0              // 0 (missing key)

// Array indexing
var arr = read_json("[10, 20, 30]", error)
let first = arr?[0] ?? -1                  // 10

// Chained access on deeply missing paths returns the default
let deep = js?.a?.b?.c ?? -1               // -1

// null pointer is safe
var nothing : JsonValue?
let safe = nothing?.foo ?? "safe"          // "safe"
```

Variant checks with is/as

`json_boost` rewrites `is` and `as` on `JsonValue?` to check the underlying `JsValue` variant:

```
var js = JV("hello")
print("{js is _string}\n")     // true
print("{js as _string}\n")     // hello

var jn = JV(42)
print("{jn is _longint}\n")    // true
```

Struct serialization

`json_boost` provides generic `JV()` and `from_JV()` that convert structs to/from JSON using compile-time reflection:

```
struct Player {
    name  : string
    hp    : int
    speed : float
}

let p = Player(name = "Hero", hp = 100, speed = 5.5)
var js = JV(p)
// {"speed":5.5, "name":"Hero", "hp":100}

var p2 = from_JV(parsed, type<Player>)
print("{p2.name}, hp={p2.hp}\n")
```

Enum serialization

Enums serialize as strings by default:

```
enum Weapon { sword; bow; staff }

var js = JV(Weapon.bow)      // "bow"
var w  = from_JV(js, type<Weapon>)   // Weapon.bow
```

Reader macro

The `%json~` reader macro embeds JSON directly in daslang code. It parses at compile time and creates a `JsonValue?` at runtime:

```
var settings = %json~
{
    "resolution": [1920, 1080],
    "fullscreen": true,
    "title": "My Game"
}
%%

let title = settings?.title ?? "untitled"   // "My Game"
```

Writer settings

Global settings control `write_json` behavior:

- `set_no_trailing_zeros(true)` — omit trailing zeros from round doubles (`1.0` → `1`)
- `set_no_empty_arrays(true)` — skip empty array fields in objects

```
let old = set_no_trailing_zeros(true)
print(write_json(JV(1.0lf)))  // 1
set_no_trailing_zeros(old)
```

Collections

JV and from_JV work with array<T> and table<string;T>:

```
var arr <- array<int>(10, 20, 30)
var js = JV(arr)           // [10, 20, 30]
var back = from_JV(js, type<array<int>>)

var tab <- { "x" => 1, "y" => 2 }
var jt = JV(tab)           // {"x":1, "y":2}
```

Vector types (float2/3/4, int2/3/4) serialize as objects with x, y, z, w keys:

```
var js = JV(float3(1.0, 2.0, 3.0))
// {"x":1, "y":2, "z":3}
```

Tuples serialize with _0, _1, … keys. Variants include a $variant field.

Broken JSON repair

try_fixing_broken_json attempts to fix common issues from LLM output:

- String concatenation: "hello" + "world" → "helloworld"
- Trailing commas: [1, 2,] → [1, 2]
- Nested quotes: "she said "hi"" → "she said \"hi\""

```
let bad = "{ \"msg\": \"hello\", }"
let fixed = try_fixing_broken_json(bad)
var js = read_json(fixed, error)
```

sprint_json

sprint_json is a builtin function that serializes any daslang value directly to a JSON string — no JsonValue? intermediate. It handles structs, classes, variants, tuples, tables, arrays, enums, pointers, and all basic types:

```
struct Record {
    id     : int
    tag    : string
    data   : Payload
    values : array<int>
    meta   : table<string; int>
    coords : tuple<int; float>
    ptr    : void?
}

var r <- Record(uninitialized
    id = 1, tag = "test",
    data = Payload(uninitialized code = 42),
    values = [1, 2, 3],
    meta <- { "x" => 10 },
    coords = (7, 3.14),
```

(continues on next page)

(continued from previous page)

```
    ptr = null
)
let compact = sprint_json(r, false)
// {"id":1,"tag":"test","data":{"code":42},"values":[1,2,3],...}

let pretty = sprint_json(r, true)
// human-readable with indentation
```

The second argument controls human-readable formatting. Works with simple values too:

```
sprint_json(42, false)          // 42
sprint_json("hello", false)     // "hello"
sprint_json([10, 20, 30], false) // [10,20,30]
```

Field annotations

Struct field annotations control how `sprint_json` serializes fields. These require `options rtti` to be enabled.

- `@optional` — skip the field if it has a default or empty value
- `@embed` — embed a string field as raw JSON (no extra quotes)
- `@unescape` — don't escape special characters in the string
- `@enum_as_int` — serialize an enum as its integer value, not a string
- `@rename="key"` — use `key` as the JSON field name instead of the daslang field name

```
struct AnnotatedConfig {
    name : string
    @optional debug : bool          // omitted when false
    @optional tags : array<string>  // omitted when empty
    @embed raw_data : string        // embedded as raw JSON
    @unescape raw_path : string     // no escaping of special chars
    pri : Priority                  // serialized as string
    @enum_as_int level : Priority   // serialized as integer
    @rename="type" _type : string   // appears as "type" in JSON
}

var c <- AnnotatedConfig(uninitialized
    name = "app", debug = false,
    raw_data = "[1,2,3]",
    raw_path = "C:\\Users\\test",
    pri = Priority.high,
    level = Priority.medium,
    _type = "widget"
)
let json_str = sprint_json(c, false)
// {"name":"app","raw_data":[1,2,3],"raw_path":"C:\Users\test","pri":"high","level":1,
↪"type":"widget"}
```

In this example: `debug` and `tags` are omitted (`@optional`), `raw_data` is embedded as `[1,2,3]` not `"[1,2,3]"` (`@embed`), `raw_path` keeps backslashes unescaped (`@unescape`), `level` is 1 instead of `"medium"` (`@enum_as_int`), and `_type` appears as `"type"` in the output (`@rename`).

@rename annotation

Use `@rename="json_key"` when the JSON key is a daslang reserved word or doesn't follow daslang naming conventions. The field keeps a safe name in code (e.g. `_type`) but serializes as the desired key. `@rename` works with `sprint_json`, `JV`, and `from_JV`:

```
struct ApiResponse {
    @rename="type" _type : string
    @rename="class" _class : int
    value : float
}

var resp = ApiResponse(_type = "widget", _class = 3, value = 1.5)
sprint_json(resp, false)
// {"type":"widget","class":3,"value":1.5}

// from_JV maps renamed keys back to struct fields
var js = read_json("{\"type\":\"button\",\"class\":5,\"value\":2.0}", error)
var result = from_JV(js, type<ApiResponse>)
// result._type == "button", result._class == 5
```

Class serialization

Both `JV`/`from_JV` and `sprint_json` work with classes. Classes serialize their fields just like structs:

```
class Animal {
    species : string
    legs : int
}

var a = new Animal(species = "cat", legs = 4)
let json_str = sprint_json(*a, false)
// {"species":"cat","legs":4}

var js = JV(*a)
print(write_json(js))
// {"legs":4,"species":"cat"}
```

See also:

Full source: `tutorials/language/30_json.das`

Next tutorial: *Regular expressions*.

Functional programming tutorial (previous tutorial).

/stdlib/json — core JSON module reference.

/stdlib/json_boost — JSON boost module reference.

5.1.31 Regular Expressions

This tutorial covers `daslib/regex` and `daslib/regex_boost` — compiling, matching, and replacing text with regular expressions in daslang.

`regex` provides the core compiler, matcher, and iterator APIs. `regex_boost` adds the `%regex~` reader macro for compile-time patterns.

Compiling and matching

`regex_compile` creates a `Regex` from a pattern string. `regex_match` returns the end position of the match (from position 0), or `-1` on failure:

```
var re <- regex_compile("hello")
let pos = regex_match(re, "hello world")   // 5
let no  = regex_match(re, "goodbye")       // -1
```

An optional third argument specifies a starting offset:

```
var re2 <- regex_compile("world")
let pos2 = regex_match(re2, "hello world", 6)   // 11
```

Character classes

Built-in shorthand classes match common character categories:

Escape	Meaning
`\w`	Word chars `[a-zA-Z0-9_]`
`\W`	Non-word chars
`\d`	Digits `[0-9]`
`\D`	Non-digits
`\s`	Whitespace (space, tab, newline, CR, form-feed, vertical-tab)
`\S`	Non-whitespace
`\t`	Tab
`\n`	Newline
`\r`	Carriage return

Example:

```
var re_num <- regex_compile("\\d+")
regex_match(re_num, "12345")    // 5

var re_ws <- regex_compile("\\s+")
regex_match(re_ws, "   x")     // 3
```

Anchors

^ anchors the match to the beginning of the string (or offset position). $ anchors to the end:

```
var re_start <- regex_compile("^hello")
regex_match(re_start, "hello")       // 5
regex_match(re_start, "say hello")   // -1

var re_full <- regex_compile("^abc$")
regex_match(re_full, "abc")          // 3
regex_match(re_full, "abcd")         // -1
```

Quantifiers

Syntax	Meaning
+	One or more (greedy)
*	Zero or more (greedy)
?	Zero or one
{n}	Exactly *n* repetitions
{n,}	*n* or more repetitions (greedy)
{n,m}	Between *n* and *m* repetitions (greedy)

```
var re_plus <- regex_compile("a+")
regex_match(re_plus, "aaa")       // 3

var re_q <- regex_compile("colou?r")
regex_match(re_q, "color")        // 5
regex_match(re_q, "colour")       // 6
```

Counted quantifiers use braces (escaped as \{ in daslang strings):

```
var re_exact <- regex_compile("\\d\{4}")
regex_match(re_exact, "1234")     // 4

var re_range <- regex_compile("a\{2,4}")
regex_match(re_range, "a")        // -1
regex_match(re_range, "aaa")      // 3
regex_match(re_range, "aaaaa")    // 4
```

Groups and alternation

Parentheses create capturing groups. | separates alternatives:

```
var re_alt <- regex_compile("cat|dog")
regex_match(re_alt, "cat")    // 3
regex_match(re_alt, "dog")    // 3
```

regex_group retrieves group captures after a successful match:

```
var re_grp <- regex_compile("(\\w+)@(\\w+)")
let inp = "user@host"
regex_match(re_grp, inp)                    // 9
print("{regex_group(re_grp, 1, inp)}\n")  // user
print("{regex_group(re_grp, 2, inp)}\n")  // host
```

Character sets

Square brackets define a set of characters to match:

- `[abc]` — matches `a`, `b`, or `c`
- `[a-z]` — matches a range
- `[^abc]` — negated set (matches anything NOT listed)
- `[\d_]` — shorthand classes work inside sets

```
var re_vowel <- regex_compile("[aeiou]+")
regex_match(re_vowel, "aeiou")    // 5

var re_neg <- regex_compile("[^0-9]+")
regex_match(re_neg, "abc")        // 3
```

Word boundaries

`\b` matches at a word boundary — the transition between `\w` and `\W` characters, or at the start/end of the string. `\B` matches at a non-boundary position:

```
var re_bnd <- regex_compile("\\bhello\\b")
regex_match(re_bnd, "hello")       // 5

var re_nb <- regex_compile("\\Bell")
regex_match(re_nb, "hello", 1)     // 4
```

Foreach and replace

`regex_foreach` iterates over all non-overlapping matches, passing each match range (as `int2`) to a block. Return `true` to continue:

```
var re_num <- regex_compile("\\d+")
regex_foreach(re_num, "a12b34c56") $(at) {
    print("[{at.x},{at.y}] ")   // [1,3] [4,6] [7,9]
    return true
}
```

`regex_replace` replaces every match using a block that receives the matched substring and returns the replacement:

```
let result = regex_replace(re_num, "a12b34c56") $(match_str) {
    return "X"
}
print("{result}\n")   // aXbXcX
```

Escaped metacharacters

Backslash escapes literal metacharacters: `\. \+ \* \( \) \[ \] \| \\ \^ \{ \}`:

```
var re_dot <- regex_compile("\\d+\\.\\d+")
regex_match(re_dot, "3.14")       // 4

var re_parens <- regex_compile("\\(\\w+\\)")
regex_match(re_parens, "(hello)")   // 7
```

Hex escapes

`\xHH` matches a character by its hexadecimal code:

```
var re_hex <- regex_compile("\\x41")
regex_match(re_hex, "A")    // 1  (0x41 = 'A')
```

Reader macro

`regex_boost` provides the `%regex~` reader macro which compiles a pattern at compile time. No double-escaping is needed — backslashes are literal in the macro body:

```
require daslib/regex_boost

var re <- %regex~\d+%%
regex_match(re, "42abc")    // 2

var re2 <- %regex~[a-z]+%%
regex_match(re2, "hello")   // 5
```

Search, split, match_all

`regex_search` finds the first match anywhere in the string (unlike `regex_match` which only matches at position 0). Returns `int2(start, end)` or `int2(-1, -1)`:

```
var re_num <- regex_compile("\\d+")
let pos = regex_search(re_num, "abc 123 def")   // int2(4, 7)
```

`regex_split` splits a string by pattern matches:

```
var re_comma <- regex_compile(",\\s*")
var parts <- regex_split(re_comma, "a, b,c, d")
// parts == ["a", "b", "c", "d"]
```

`regex_match_all` collects all match ranges:

```
var re_word <- regex_compile("\\w+")
var matches <- regex_match_all(re_word, "foo bar baz")
// length(matches) == 3
```

Non-capturing groups

`(?:...)` groups without creating a capture. Useful for applying quantifiers or alternation without increasing the group count:

```
var re <- regex_compile("(?:cat|dog)fish")
regex_match(re, "catfish")     // 7
length(re.groups)              // 1 (only group 0)

var re2 <- regex_compile("(?:ab)\{3}")
regex_match(re2, "ababab")     // 6
```

Named groups

`(?P<name>...)` creates a named capturing group, accessible via `regex_group_by_name`:

```
var re <- regex_compile("(?P<user>\\w+)@(?P<host>\\w+)")
let email = "alice@example"
regex_match(re, email)
regex_group_by_name(re, "user", email)   // "alice"
regex_group_by_name(re, "host", email)   // "example"
```

Named groups are also accessible by their numeric index (1, 2, …) via `regex_group`.

Lazy quantifiers

Greedy quantifiers (+, *, ?, `{n,m}`) match as much as possible. Appending ? makes them lazy — matching as little as possible:

Greedy	Lazy	Meaning
+	+?	One or more (prefer fewer)
*	*?	Zero or more (prefer fewer)
?	??	Zero or one (prefer zero)
`{n,m}`	`{n,m}?`	Counted (prefer *n*)

```
// lazy +? takes the shortest match
var re <- regex_compile("<.+?>")
let pos = regex_search(re, "<b>bold</b>")
// pos == int2(0, 3) - matches "<b>" not "<b>bold</b>"

// greedy vs lazy at end of pattern
regex_match(regex_compile("a+"),  "aaa")   // 3 (greedy: all)
regex_match(regex_compile("a+?"), "aaa")   // 1 (lazy: one)
```

Practical examples

The `%regex~` reader macro avoids double-escaping, making real-world patterns much more readable.

Phone number validation:

```
var re_phone <- %regex~^\d{3}-\d{4}$%%
regex_match(re_phone, "555-1234") != -1   // true
regex_match(re_phone, "55-1234")  != -1   // false
```

Strip non-word characters:

```
var re_strip <- %regex~[^\w]+%%
let cleaned = regex_replace(re_strip, "he!l@l#o") $(m) {
    return ""
}
// cleaned == "hello"
```

Extract email parts:

```
var re_email <- %regex~([\w.]+)@([\w.]+)%%
let email = "user@example.com"
regex_match(re_email, email)
regex_group(re_email, 1, email)   // "user"
regex_group(re_email, 2, email)   // "example.com"
```

IP address pattern:

```
var re_ip <- %regex~\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}%%
regex_match(re_ip, "192.168.1.1")   // 11
```

Case-insensitive matching

Pass `case_insensitive=true` to `regex_compile` for ASCII case-insensitive matching. Character classes and sets are also affected:

```
// default: case-sensitive
var re <- regex_compile("hello")
regex_match(re, "HELLO")    // -1

// case-insensitive
var re_ci <- regex_compile("hello", [case_insensitive=true])
regex_match(re_ci, "HELLO")    // 5
regex_match(re_ci, "HeLLo")   // 5

// character sets are also case-insensitive
var re_set <- regex_compile("[a-z]+", [case_insensitive=true])
regex_match(re_set, "AbCdE")   // 5
```

Dot and newline

By default, `.` matches any character **except** newline (\n). Pass `dot_all=true` to `regex_compile` to make `.` match newlines too:

```
// default: '.' does NOT match newline
var re <- regex_compile(".+")
regex_match(re, "ab\nc")     // 2

// dot_all=true: '.' also matches newline
var re_all <- regex_compile(".+", [dot_all=true])
regex_match(re_all, "ab\nc")   // 4
```

This is useful for multi-line content extraction:

```
var re <- regex_compile("START(.+?)END", [dot_all=true])
let text = "START\nhello\nEND"
regex_match(re, text)
regex_group(re, 1, text)   // "\nhello\n"
```

Lookahead assertions

Lookahead assertions check what follows the current position without consuming any input.

`(?=...)` is a **positive lookahead** — the overall match succeeds only if the lookahead pattern matches:

```
// "foo" only if followed by "bar"
var re <- regex_compile("foo(?=bar)")
regex_match(re, "foobar")   // 3 (matches "foo", not "foobar")
regex_match(re, "foobaz")   // -1

// extract digits before " dollars"
var re2 <- regex_compile("\\d+(?= dollars)")
let pos = regex_search(re2, "100 dollars")
// pos == int2(0, 3) - matches "100"
```

`(?!...)` is a **negative lookahead** — the match succeeds only if the lookahead pattern does NOT match:

```
// "foo" only if NOT followed by "bar"
var re <- regex_compile("foo(?!bar)")
regex_match(re, "foobar")   // -1
regex_match(re, "foobaz")   // 3

// single char NOT followed by "!"
var re2 <- regex_compile("\\w(?!!)")
regex_match(re2, "a!")   // -1
regex_match(re2, "a.")   // 1
```

Template-string replace

`regex_replace` also accepts a replacement template string instead of a block. The template supports group references:

Reference	Meaning
`$0`	Whole match
`$&`	Whole match (alternative syntax)
`$1`–`$9`	Numbered capturing groups
`${name}`	Named capturing group
`$$`	Literal `$` character

```
// swap first and last name
var re <- regex_compile("(\\w+) (\\w+)")
regex_replace(re, "John Smith", "$2 $1")     // "Smith John"

// wrap each word in brackets
var re2 <- regex_compile("\\w+")
regex_replace(re2, "hello world", "[$0]")    // "[hello] [world]"
```

Named group references use `${name}` syntax:

```
var re <- regex_compile("(?P<m>\\d+)/(?P<d>\\d+)/(?P<y>\\d+)")
regex_replace(re, "12/25/2024", "${y}-${m}-${d}")   // "2024-12-25"
```

Reader macro flags

Flags can be appended after a second `~` in the `%regex~` reader macro:

Syntax	Effect
`%regex~pattern~i%%`	Case-insensitive matching
`%regex~pattern~s%%`	Dot-all mode (`.` matches `\n`)
`%regex~pattern~is%%`	Both flags combined

```
var re_ci <- %regex~hello~i%%
regex_match(re_ci, "HELLO")   // 5

var re_s <- %regex~.+~s%%
regex_match(re_s, "ab\nc")    // 4

var re_is <- %regex~hello.+world~is%%
regex_match(re_is, "Hello\nWorld")   // 11
```

See also:

Full source: `tutorials/language/31_regex.das`

JSON tutorial (previous tutorial).

Next tutorial: *Operator overloading*.

/stdlib/regex — core regex module reference.

/stdlib/regex_boost — regex boost module reference.

5.1.32 Operator Overloading

This tutorial covers operator overloading in daslang — defining custom behavior for built-in operators when applied to your own types.

daslang operators are regular functions named `def operator <op>` and follow standard overload-resolution rules. They can be declared as free functions or as struct methods.

Arithmetic operators

Overload +, -, *, /, % to give your types arithmetic behaviour. Each operator is a binary function that takes two values and returns a result:

```
struct Vec2 {
    x : float
    y : float
}

def operator +(a, b : Vec2) : Vec2 {
    return Vec2(x = a.x + b.x, y = a.y + b.y)
}

def operator -(a, b : Vec2) : Vec2 {
    return Vec2(x = a.x - b.x, y = a.y - b.y)
}

def operator *(a : Vec2; s : float) : Vec2 {
    return Vec2(x = a.x * s, y = a.y * s)
}

def operator *(s : float; a : Vec2) : Vec2 {
    return a * s
}

def operator /(a : Vec2; s : float) : Vec2 {
    return Vec2(x = a.x / s, y = a.y / s)
}
```

Usage:

```
let a = Vec2(x = 1.0, y = 2.0)
let b = Vec2(x = 3.0, y = 4.0)
print("{(a + b).x}, {(a + b).y}\n")  // 4, 6
print("{(a * 3.0).x}\n")             // 3
```

Note that `operator *` is defined twice — once for `Vec2 * float` and once for `float * Vec2` — because daslang does not automatically commute arguments.

Comparison operators

Overload ==, !=, <, >, <=, >= for custom comparisons. Returning `bool` is required:

```
def operator ==(a, b : Vec2) : bool {
    return a.x == b.x && a.y == b.y
}

def operator !=(a, b : Vec2) : bool {
    return !(a == b)
}

// Order by squared magnitude (avoids sqrt)
def operator <(a, b : Vec2) : bool {
    return (a.x * a.x + a.y * a.y) < (b.x * b.x + b.y * b.y)
}
```

Usage:

```
let a = Vec2(x = 1.0, y = 2.0)
let c = Vec2(x = 3.0, y = 4.0)
print("a < c: {a < c}\n")   // true
```

Note: daslang does **not** auto-generate != from ==, or > from <. Each comparison operator must be defined explicitly if needed.

Unary operators

Overload unary - (negate), ~ (bitwise complement), ! (logical not), and prefix/postfix ++/--:

```
def operator -(a : Vec2) : Vec2 {
    return Vec2(x = -a.x, y = -a.y)
}
```

Usage:

```
let a = Vec2(x = 1.0, y = 2.0)
let neg = -a
print("-a = ({neg.x}, {neg.y})\n")   // -a = (-1, -2)
```

For increment/decrement, ++`operator` is prefix (modify-then-use) while `operator` ++ is postfix (use-then-modify).

Compound assignment operators

Overload +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= and others for in-place modification. The first argument must be `var ... &` (mutable reference):

```
def operator +=(var a : Vec2&; b : Vec2) {
    a.x += b.x
    a.y += b.y
}

def operator -=(var a : Vec2&; b : Vec2) {
    a.x -= b.x
    a.y -= b.y
}

def operator *=(var a : Vec2&; s : float) {
    a.x *= s
    a.y *= s
}
```

Usage:

```
var a = Vec2(x = 1.0, y = 2.0)
a += Vec2(x = 0.5, y = 0.5)
// a is now (1.5, 2.5)
```

Index operators

Index operators control how `[]` behaves on your types:

Operator	Purpose
`operator []`	Read access — `v = obj[i]`
`operator []=`	Write access — `obj[i] = v`
`operator []+=`	Compound index — `obj[i] += v`
`operator []-=`	Compound index — `obj[i] -= v`
`operator []<-`	Move into index — `obj[i] <- v`
`operator []:=`	Clone into index — `obj[i] := v`

Example as free functions:

```
struct Matrix2x2 {
    data : float[4]
}

def operator [](m : Matrix2x2; i : int) : float {
    return m.data[i]
}

def operator []=(var m : Matrix2x2&; i : int; v : float) {
    m.data[i] = v
}
```

(continues on next page)

(continued from previous page)

```
def operator []+=(var m : Matrix2x2&; i : int; v : float) {
    m.data[i] += v
}
```

Usage:

```
var m : Matrix2x2
m.data[0] = 1.0
m[0] = 10.0       // calls operator []=
m[0] += 5.0        // calls operator []+=
print("{m[0]}\n")  // 15
```

The safe index operator `?[]` returns a default value when the index is out of range, following the same pattern.

Dot operators / property accessors

Dot operators let you create computed properties that look like regular field access.

`operator . name` defines a getter, `operator . name :=` defines a setter:

```
struct Particle {
    pos_x : float
    pos_y : float
}

// getter - computed "speed" property
def operator . speed(p : Particle) : float {
    return sqrt(p.pos_x * p.pos_x + p.pos_y * p.pos_y)
}

// setter - scales direction to target speed
def operator . speed := (var p : Particle&; value : float) {
    let mag = sqrt(p.pos_x * p.pos_x + p.pos_y * p.pos_y)
    if (mag > 0.0) {
        let scale = value / mag
        p.pos_x *= scale
        p.pos_y *= scale
    }
}
```

Usage:

```
var p = Particle(pos_x = 3.0, pos_y = 4.0)
print("{p.speed}\n")  // 5
p.speed := 10.0
// p is now (6, 8)
```

The generic `operator .` takes a string field name and intercepts all field accesses. This is powerful but should be used sparingly:

```
def operator .(t : MyType; name : string) : string {
    return "accessing: {name}"
}
```

The ". ." (dot-space-dot) syntax bypasses any overloaded dot operator to access real struct fields:

```
print("{p . . pos_x}\n")   // accesses the actual field
```

Clone and finalize

`operator :=` overloads clone behaviour (the `:=` operator) and `operator delete` (or defining a `finalize` function) overloads deletion:

```
struct Resource {
    name : string
    refcount : int
}

def operator :=(var dst : Resource&; src : Resource) {
    dst.name = src.name
    dst.refcount = src.refcount + 1
}
```

Usage:

```
var orig = Resource(name = "texture", refcount = 1)
var copy : Resource
copy := orig
print("{copy.refcount}\n")   // 2 - clone incremented the count
```

See also:

Move, copy, and clone for more details.

Struct method operators

Operators can be defined as struct methods instead of free functions. This is useful for encapsulation — the operator lives with the type definition:

```
struct Stack {
    items : array<int>

    def const operator [](index : int) : int {
        return items[index]
    }

    def operator []=(index : int; value : int) {
        items[index] = value
    }
}
```

The `const` qualifier on the getter means it does not modify the struct. Write operators omit `const` because they mutate state.

Usage:

```
var s : Stack
s.items |> push(10)
s.items |> push(20)
print("{s[0]}\n")   // 10
s[1] = 99
print("{s[1]}\n")   // 99
```

Generic operators

Operators (and operator-like functions) can be generic using `auto` types. This lets one definition cover multiple types, as long as they have the required fields:

```
struct Vec3 {
    x : float
    y : float
    z : float
}

def dot_2d(a, b : auto) : float {
    return a.x * b.x + a.y * b.y
}
```

Both `Vec2` and `Vec3` have `x` and `y` fields, so `dot_2d` works with either:

```
let v2a = Vec2(x = 1.0, y = 0.0)
let v2b = Vec2(x = 0.0, y = 1.0)
print("{dot_2d(v2a, v2b)}\n")   // 0

let v3a = Vec3(x = 3.0, y = 4.0, z = 0.0)
let v3b = Vec3(x = 1.0, y = 0.0, z = 99.0)
print("{dot_2d(v3a, v3b)}\n")   // 3
```

For more sophisticated constraints, use *contracts*.

Complete operator reference

The full list of overloadable operators in daslang:

Category	Operators
Arithmetic	`+` `-` `*` `/` `%`
Comparison	`==` `!=` `<` `>` `<=` `>=`
Bitwise	`&` `|` `^` `~` `<<` `>>` `<<<` `>>>`
Logical	`&&` `||` `^^` `!`
Unary	`-` (negate) `~` (complement) `++` `--`
Compound assignment	`+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=` `<<<=` `>>>=` `&&=` `||=` `^^=`
Index	`[]` `[]=` `[]<-` `[]:=` `[]+=` `[]-=` `[]*=` etc.
Safe index	`?[]`
Dot	`.` `?.` `.` `name` `.` `name` `:=` `.` `name` `+=` etc.
Type	`:=` (clone) `delete` (finalize) `is` `as` `?as`
Null coalesce	`??`
Interval	`..`

See also:

Full source: `tutorials/language/32_operator_overloading.das`

Next tutorial: *Algorithm*

Regular expressions tutorial (previous tutorial).

Functions — function declaration and operator overloading reference.

Generic programming — generic operators with `auto` types.

5.1.33 Algorithm

This tutorial covers the `algorithm` module — a collection of general-purpose algorithms for searching, sorting, manipulating arrays, and performing set operations on tables. All functions live in `daslib/algorithm` and many also support fixed-size arrays via `[expect_any_array]` overloads.

```
require daslib/algorithm
```

Binary search family

The module provides the classic binary search family for sorted arrays:

- `lower_bound` — first element `>= val`
- `upper_bound` — first element `> val`
- `binary_search` — `true` if `val` exists
- `equal_range` — `int2(lower_bound, upper_bound)`

Each comes with overloads for full or sub-range search and custom comparators:

```
var a <- [1, 2, 2, 2, 3, 5, 8, 13]
print("lower_bound(2) = {lower_bound(a, 2)}\n")    // 1
print("upper_bound(2) = {upper_bound(a, 2)}\n")    // 4
print("equal_range(2)  = {equal_range(a, 2)}\n")   // (1, 4)
print("binary_search(4) = {binary_search(a, 4)}\n") // false
```

Custom comparators let you search in non-default order:

```
var desc <- [9, 7, 5, 3, 1]
let idx = lower_bound(desc, 5) $(lhs, rhs : int const) {
    return lhs > rhs
}
print("{idx}\n") // 2
```

Sorting and deduplication

`sort_unique` sorts an array in place and removes duplicates. `unique` removes only *adjacent* duplicates (like C++ `std::unique`), so the array should be sorted first for full deduplication:

```
var a <- [5, 3, 1, 3, 5, 1, 2]
sort_unique(a)
print("{a}\n")  // [1, 2, 3, 5]

var b <- [3, 1, 3, 1]
var c <- unique(b)
print("{c}\n")  // [3, 1, 3, 1] - no adjacent dups removed
```

Array manipulation

```
// reverse - in place
var a <- [1, 2, 3, 4, 5]
reverse(a)          // [5, 4, 3, 2, 1]

// combine - concatenate into a new array
var both <- combine([1, 2], [3, 4])  // [1, 2, 3, 4]

// fill - set all elements
fill(a, 0)          // [0, 0, 0, 0, 0]

// rotate - mid becomes first
var b <- [1, 2, 3, 4, 5]
rotate(b, 2)        // [3, 4, 5, 1, 2]

// erase_all - remove all occurrences of a value
var c <- [1, 2, 3, 2, 4]
erase_all(c, 2)     // [1, 3, 4]
```

Querying arrays

```
var a <- [3, 1, 4, 1, 5, 9, 2, 6]
print("is_sorted: {is_sorted(a)}\n")       // false
print("min index: {min_element(a)}\n")      // 1 (value 1)
print("max index: {max_element(a)}\n")      // 5 (value 9)
```

`min_element` and `max_element` return the *index* (not the value) of the minimum/maximum element, or `-1` for empty arrays. Both accept optional custom comparators.

Set operations

Tables with no value type serve as sets. The module provides:

```
var a <- { 1, 2, 3, 4 }
var b <- { 3, 4, 5, 6 }

var inter <- intersection(a, b)           // {3, 4}
var uni   <- union(a, b)                  // {1, 2, 3, 4, 5, 6}
var diff  <- difference(a, b)             // {1, 2}
var sdiff <- symmetric_difference(a, b)   // {1, 2, 5, 6}

print("identical: {identical(a, a)}\n")   // true
print("is_subset({2, 3}, a): {is_subset({2, 3}, a)}\n") // true
```

Topological sort

`topological_sort` orders nodes respecting dependency edges. Each node must have an `id` field and a `before` table listing which ids must come first:

```
struct TsNode {
    id     : int
    before : table<int>
}

var nodes <- [TsNode(
    id=2, before <- {1}), TsNode(
    id=1, before <- {0}), TsNode(
    id=0
)]
var sorted <- topological_sort(nodes)
// sorted: [0, 1, 2]
```

If the graph contains a cycle, `topological_sort` calls `panic`.

Fixed-size arrays

Most functions (`reverse`, `fill`, `lower_bound`, `binary_search`, `upper_bound`, `min_element`, `max_element`, `is_sorted`, `rotate`, `erase_all`, `combine`) also work on fixed-size arrays:

```
var a = fixed_array<int>(5, 3, 1, 4, 2)
print("min: {min_element(a)}\n")    // 2 (value 1)
reverse(a)                          // [2, 4, 1, 3, 5]
```

See also:

Full source: `tutorials/language/33_algorithm.das`

Previous tutorial: *Operator Overloading*

Next tutorial: *Entity Component System (DECS)*

Arrays — array language reference.

Tables — table language reference.

5.1.34 Entity Component System (DECS)

This tutorial covers the `decs` module — daslang's built-in Entity Component System. DECS provides a lightweight ECS where entities are identified by `EntityId`, carry dynamically typed components, and are grouped into archetypes based on their component sets. All mutations are deferred and applied on `commit()`.

```
require daslib/decs_boost
```

Core concepts

- **Entity** — an `EntityId` with associated component data.
- **Component** — a named, typed value attached to an entity (set via `:=` on a `ComponentMap`).
- **Archetype** — a storage bucket for entities with the same set of component names. Created automatically.
- **Deferred execution** — `create_entity`, `delete_entity`, and `update_entity` are all deferred; call `commit()` to apply them.

Creating entities

`create_entity` takes a block that receives the `EntityId` and a `ComponentMap`. Use `:=` to set components:

```
let player = create_entity() @(eid, cmp) {
    cmp.name := "hero"
    cmp.hp   := 100
    cmp.pos  := float3(0, 0, 0)
}
commit()  // entity becomes visible
```

Querying entities

Global queries iterate all entities matching the listed component types. Component names in the block signature match component names on entities:

```
query() $(name : string; hp : int; pos : float3) {
    print("  {name}: hp={hp} pos={pos}\n")
}
```

Query a specific entity by passing its `EntityId`:

```
query(eid) $(tag : string; val : int) {
    print("Found: tag={tag} val={val}\n")
}
```

Mutable queries

Use `var` and `&` to modify components in place:

```
query() $(var pos : float3&; vel : float3) {
    pos += vel
}
```

REQUIRE and REQUIRE_NOT

Filter queries to entities that have (or lack) specific components, even when you don't need those components as block arguments:

```
// Only entities WITH a "weapon" component
query <| $ [REQUIRE(weapon)] (name : string; hp : int) {
    print("  {name} hp={hp}\n")
}

// Exclude entities WITH a "shield" component
query <| $ [REQUIRE_NOT(shield)] (name : string) {
    print("  {name}\n")
}
```

find_query

`find_query` stops iteration when the block returns `true`, providing an early-exit search:

```
let found = find_query() $(idx : int) {
    if (idx == 7) {
        return true
    }
    return false
}
```

Deleting entities

`delete_entity` is deferred until `commit()`:

```
delete_entity(eid)
commit()
```

Updating entities

`update_entity` lets you modify, add, or remove components. If the component set changes, the entity moves to a different archetype:

```
update_entity(eid) @(eid, cmp) {
    var hp = 0
    hp = get(cmp, "hp", hp)
    cmp |> set("hp", hp - 25)
    cmp.enraged := true
}
commit()

// Remove a component
update_entity(eid) @(eid, cmp) {
    cmp |> remove("enraged")
}
commit()
```

Default values in queries

If an entity lacks a queried component, the default value is used. Parameters with defaults must be `const` (no `var`, no `&`):

```
query() $(name : string; alpha : float = 0.5) {
    print("  {name}: alpha={alpha}\n")
}
```

Templates

`[decs_template]` structs map struct fields to components with an automatic prefix (`StructName_` by default). This generates `apply_decs_template` and `remove_decs_template` functions:

```
[decs_template]
struct Particle {
    pos  : float3
    vel  : float3
    life : int
}

// Create entity with template
create_entity() @(eid, cmp) {
    apply_decs_template(cmp, Particle(
        pos  = float3(0, 0, 0),
        vel  = float3(1, 0, 0),
        life = 100
    ))
}

// Query using template struct
query() $(var p : Particle) {
```

(continues on next page)

(continued from previous page)

```
    p.pos += p.vel
    p.life -= 1
}
```

Stage functions

Stage functions are annotated with `[decs(stage=name)]` and become queries that run when you call `decs_stage("name")`. The stage commits automatically before and after running all registered functions:

```
[decs(stage = simulate)]
def simulate_particles(var p : Particle) {
    p.pos += p.vel
    p.life -= 1
}

// Run 3 simulation steps
for (step in range(3)) {
    decs_stage("simulate")
}
```

Nested queries

Queries can be nested — the inner query sees all matching entities:

```
query() $(name : string) {
    var sum = 0
    query() $(val : int) {
        sum += val
    }
    print("  {name} sees total val={sum}\n")
}
```

Serialization

The entire ECS state can be saved and restored using the `archive` module:

```
require daslib/archive

var data <- mem_archive_save(decsState)
restart()
mem_archive_load(data, decsState)
```

Entity ID recycling

When an entity is deleted, its ID slot is recycled. The generation counter increments so that stale `EntityId` values cannot accidentally access the new entity occupying the same slot:

```
let eid1 = create_entity() @(eid, cmp) {
    cmp.val := 1
}
commit()
delete_entity(eid1)
commit()

let eid2 = create_entity() @(eid, cmp) {
    cmp.val := 2
}
commit()
// eid2.id == eid1.id but eid2.generation != eid1.generation
```

Archetype inspection

You can inspect the world state through `decsState`:

```
print("Archetypes: {length(decsState.allArchetypes)}\n")
for (arch in decsState.allArchetypes) {
    print("  {arch.size} entities, components:")
    for (c in arch.components) {
        print(" {c.name}")
    }
    print("\n")
}
```

Utility functions

`is_alive` checks whether an `EntityId` still refers to a living entity. It returns `false` for `INVALID_ENTITY_ID`, deleted entities, and stale generation IDs after slot recycling:

```
print("alive? {is_alive(hero)}\n")        // true
delete_entity(hero)
commit()
print("alive? {is_alive(hero)}\n")        // false
```

`entity_count` returns the total number of alive entities across all archetypes:

```
print("entity count: {entity_count()}\n")
```

`get_component` retrieves a single component value by entity ID and name. The type is inferred from the default value parameter. If the entity is dead or the component is not present, the default is returned:

```
let hp = get_component(hero, "hp", 0)             // returns int
let pos = get_component(hero, "pos", float3(0))   // returns float3
let missing = get_component(hero, "shield", -1)    // returns -1 (not found)
let dead = get_component(deleted_eid, "hp", -999)  // returns -999 (dead)
```

See also:

Full source: `tutorials/language/34_decs.das`

Previous tutorial: *Algorithm*

Next tutorial: *Job Queue (jobque)*

Structures — struct language reference.

Iterators — iterator language reference.

Lambda — lambda language reference.

5.1.35 Job Queue (jobque)

This tutorial covers the `jobque_boost` module — daslang's built-in multi-threading primitives. The module provides a work-stealing thread pool, typed channels for inter-thread communication, job statuses and wait groups for synchronization, lock boxes for shared mutable state, and high-level helpers like `parallel_for`.

```
require daslib/jobque_boost
```

Core concepts

- **Job queue** — a work-stealing thread pool initialized by `with_job_que`. All multi-threading operations must be performed inside this scope.
- **Job** — a unit of work dispatched to the thread pool via `new_job`. Each job runs in a *cloned* context — it does not share memory with the caller.
- **Channel** — a thread-safe FIFO queue for passing typed struct data between jobs. Created with `with_channel(count)` where `count` is the expected number of `notify_and_release` calls before the channel closes.
- **JobStatus** — a counter that tracks asynchronous completion. `notify_and_release` decrements it, `join` blocks until it reaches zero.
- **Wait group** — `with_wait_group` wraps a `JobStatus` with automatic `join` on scope exit. `done` is an alias for `notify_and_release`.
- **LockBox** — a mutex-protected single-value container. `set` stores a value, `get` reads it inside a block.
- **Thread** — `new_thread` spawns a dedicated OS thread, outside the thread pool.

Channel and lock-box data **must be a struct** (or handled type) — primitives cannot be heap-allocated. Wrap them in a struct when needed.

Starting the job queue

`with_job_que` initializes the thread pool and runs the block. All job-queue operations must happen inside this scope:

```
with_job_que() {
    print("Job queue is active\n")
}
// output: Job queue is active
```

Spawning jobs

`new_job` dispatches work to the thread pool. Each job runs in a cloned context. Use channels to communicate results back:

```
with_job_que() {
    with_channel(1) $(ch) {
        new_job() @() {
            ch |> push_clone(IntVal(v = 42))
            ch |> notify_and_release
        }
        ch |> for_each_clone() $(val : IntVal#) {
            print("Received from job: {val.v}\n")
        }
    }
}
// output: Received from job: 42
```

Channels

A `Channel` is a thread-safe FIFO queue. Create one with `with_channel(expected_count)` where the count matches the number of `notify_and_release` calls that will be made. `push_clone` sends data and `for_each_clone` receives it, blocking until the channel is drained:

```
with_job_que() {
    with_channel(2) $(ch) {
        new_job() @() {
            ch |> push_clone(StringVal(s = "hello"))
            ch |> notify_and_release
        }
        new_job() @() {
            ch |> push_clone(StringVal(s = "world"))
            ch |> notify_and_release
        }
        var messages : array<string>
        ch |> for_each_clone() $(msg : StringVal#) {
            messages |> push(clone_string(msg.s))
        }
        sort(messages)
        for (m in messages) {
            print("  {m}\n")
        }
    }
}
// output:
//   hello
//   world
```

Channels work with any struct type — not just simple wrappers:

```
struct WorkResult {
    index : int
    value : int
```

(continues on next page)

(continued from previous page)

```
}

with_channel(1) $(ch) {
    new_job() @() {
        for (i in range(3)) {
            ch |> push_clone(WorkResult(index = i, value = i * i))
        }
        ch |> notify_and_release
    }
    ch |> for_each_clone() $(r : WorkResult#) {
        print("  [{r.index}] = {r.value}\n")
    }
}
// output:
//   [0] = 0
//   [1] = 1
//   [2] = 4
```

Multiple producers

When multiple jobs push to the same channel, set the channel count to the number of producers. `for_each_clone` blocks until all have called `notify_and_release`:

```
let num_producers = 3
with_channel(num_producers) $(ch) {
    for (p in range(num_producers)) {
        new_job() @() {
            ch |> push_clone(IntVal(v = p * 100))
            ch |> notify_and_release
        }
    }
    var results : array<int>
    ch |> for_each_clone() $(val : IntVal#) {
        results |> push(val.v)
    }
    sort(results)
    print("producers: {results}\n")
}
// output: producers: [0, 100, 200]
```

JobStatus

A `JobStatus` tracks completion of asynchronous work. `with_job_status(count)` creates a status expecting `count` notifications. Each call to `notify_and_release` decrements the counter. `join` blocks until all notifications arrive:

```
with_job_que() {
    with_job_status(3) $(status) {
        for (i in range(3)) {
            new_job() @() {
                status |> notify_and_release
```

(continues on next page)

(continued from previous page)

```
            }
        }
        status |> join
    }
    print("All 3 jobs finished\n")
}
// output: All 3 jobs finished
```

Wait groups

`with_wait_group` is a convenience wrapper — it combines `with_job_status` and `join`. `done` is an alias for `notify_and_release`:

```
with_job_que() {
    with_wait_group(3) $(wg) {
        for (i in range(3)) {
            new_job() @() {
                wg |> done
            }
        }
    }
    print("all jobs complete\n")
}
// output: all jobs complete
```

LockBox

A `LockBox` holds a single struct value protected by a mutex. `set` stores a value and `get` reads it inside a block. These operations must run inside a job context (`new_job` or `new_thread`):

```
struct BoxCounter {
    value : int
}

with_job_que() {
    with_lock_box() $(box) {
        with_channel(1) $(ch) {
            new_job() @() {
                box |> set(BoxCounter(value = 42))
                box |> get() $(c : BoxCounter#) {
                    ch |> push_clone(IntVal(v = c.value))
                }
                box |> release
                ch |> notify_and_release
            }
            ch |> for_each_clone() $(val : IntVal#) {
                print("lockbox value: {val.v}\n")
            }
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
// output: lockbox value: 42
```

parallel_for

`parallel_for` splits a range `[begin..end)` into chunks and runs them on the job queue. A macro automatically wraps the block body in `new_job` — you write sequential code inside:

```
with_job_que() {
    let num_jobs = 3
    with_channel(num_jobs) $(ch) {
        parallel_for(0, 10, num_jobs) $(job_begin, job_end, wg) {
            for (i in range(job_begin, job_end)) {
                ch |> push_clone(IntVal(v = i * i))
            }
            ch |> notify_and_release
            wg |> done
        }
        var results : array<int>
        ch |> for_each_clone() $(val : IntVal#) {
            results |> push(val.v)
        }
        sort(results)
        print("squares: {results}\n")
    }
}
// output: squares: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The block receives three parameters:

- `job_begin`, `job_end` — the index sub-range for this chunk
- `wg` — a `JobStatus?` that must be signaled via `done(wg)` when the chunk finishes

When using a channel inside `parallel_for`, set the channel count to `num_jobs` since each chunk calls `notify_and_release` once.

new_thread

`new_thread` creates a dedicated OS thread outside the thread pool. Use it for long-lived work that should not block the job queue:

```
with_job_que() {
    with_channel(1) $(ch) {
        new_thread() @() {
            ch |> push_clone(StringVal(s = "from thread"))
            ch |> notify_and_release
        }
        ch |> for_each_clone() $(msg : StringVal#) {
            print("{msg.s}\n")
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
// output: from thread
```

See also:

Full source: `tutorials/language/35_jobque.das`

Previous tutorial: *Entity Component System (DECS)*

Next tutorial: *Pointers*

Lambda — lambda language reference.

Blocks — block language reference.

5.1.36 Pointers

This tutorial covers pointer types, creation, dereferencing, null safety, pointer arithmetic, and low-level operations like `reinterpret` and `intptr`.

Pointer types

In daslang, `T?` is a nullable pointer to `T`:

```
var p : int?           // pointer to int - null by default
var ps : Point?        // pointer to struct
var vp : void?         // untyped (void) pointer
```

Pointers are used for heap-allocated data, optional references, and low-level interop. Unlike C/C++, field access auto-dereferences:

```
p.x      // same as (*p).x - no -> operator needed
```

Creating pointers with new

`new` allocates on the heap and returns a pointer:

```
var p = new Point(x = 3.0, y = 4.0)  // p is Point?
```

Fields get default values when omitted:

```
var q = new Point()     // x = 0.0, y = 0.0
```

Heap pointers must be freed explicitly with `delete` (requires `unsafe`), or use `var inscope` for automatic cleanup:

```
var inscope pt = new Point(x = 1.0, y = 2.0)
// pt is automatically deleted when it goes out of scope
```

addr and safe_addr

addr(x) returns a pointer to an existing variable. It requires unsafe because the pointer could outlive the variable:

```
var x = 42
unsafe {
    var p = addr(x)    // p is int?
    *p = 100           // modifies x through the pointer
}
```

safe_addr from daslib/safe_addr returns a temporary pointer (T?#) without requiring unsafe:

```
require daslib/safe_addr
var a = 13
var p = safe_addr(a)   // p is int?# - safe, no unsafe needed
print("{*p}\n")
```

Dereferencing

*p and deref(p) follow the pointer to the value. Both panic if the pointer is null:

```
unsafe {
    var y = 7
    var p = addr(y)
    print("{*p}\n")         // 7
    print("{deref(p)}\n")   // 7
}
```

For struct pointers, . auto-dereferences — no -> operator:

```
var inscope pt = new Point(x = 5.0, y = 6.0)
print("{pt.x}\n")           // 5 - same as (*pt).x
```

Null safety

Uninitialized pointers are null. Dereferencing null panics:

```
var np : int?       // null
try {
    print("{*np}\n")
} recover {
    print("null deref caught\n")
}
```

Use null checks, safe navigation, or null coalescing to handle nullable pointers safely:

```
options gen2

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

(continues on next page)

(continued from previous page)

```
def get_x(p : Point?) : float {
    return p?.x ?? -1.0    // -1.0 if p is null
}
```

`?.` returns null if the pointer is null (no panic). `??` provides a fallback value when the left side is null.

Passing pointers to functions

Pointers can be passed as function arguments. Functions can read from and write through them:

```
def double_value(p : int?) {
    *p = *p + *p
}

var val = 21
unsafe {
    double_value(addr(val))
}
print("{val}\n")  // 42
```

Struct pointer arguments auto-deref for field access:

```
def move_point(p : Point?; dx : float; dy : float) {
    p.x += dx    // auto-deref
    p.y += dy
}
```

Deletion

`delete` frees heap memory and sets the pointer to null. Requires `unsafe`:

```
var p = new Point(x = 1.0, y = 2.0)
unsafe {
    delete p     // memory freed, p is now null
}
```

Prefer `var inscope` over manual `delete` — it adds a `finally` block that automatically cleans up the pointer when the scope exits.

Pointer arithmetic

Pointer indexing and arithmetic are unsafe. They operate on raw memory with no bounds checking:

```
var arr <- [10, 20, 30, 40, 50]
unsafe {
    var p = addr(arr[0])
    print("{p[0]}, {p[2]}\n")     // 10, 30

    ++ p            // advance by one element
```

(continues on next page)

(continued from previous page)

```
    print("{*p}\n") // 20

    p += 2           // advance by two more
    print("{*p}\n") // 40
}
```

Warning: Pointer arithmetic can easily cause out-of-bounds access. No runtime bounds checking is performed.

void pointers

`void?` is an untyped pointer — equivalent to `void*` in C. Used for C/C++ interop where the actual type is opaque. Must `reinterpret` back to a typed pointer before use:

```
unsafe {
    var x = 123
    var px = addr(x)
    var vp : void? = reinterpret<void?> px   // erase type
    var px2 = reinterpret<int?> vp           // restore type
    print("{*px2}\n")                        // 123
}
```

intptr

`intptr(p)` converts a pointer to a `uint64` integer representing its memory address. Useful for debugging, logging, or identity comparisons:

```
unsafe {
    var x = 42
    var p = addr(x)
    print("address: {intptr(p)}\n")
}
```

reinterpret

`reinterpret<T>` performs a raw bit cast between types of the same size. Requires `unsafe`:

```
unsafe {
    let f = 1.0
    let bits = reinterpret<int> f    // IEEE 754: 0x3f800000
    let back = reinterpret<float> bits
    print("{back}\n")                // 1
}
```

Warning: `reinterpret` does not convert values — it reinterprets raw bits. The source and target types must have the same size.

See also:

Pointers — pointer language reference.

Unsafe — unsafe operations reference.

Values and Data Types — all data types including smart pointers.

Full source: `tutorials/language/36_pointers.das`

Previous tutorial: *Job Queue (jobque)*

Next tutorial: *Utility Patterns (defer + static_let)*

5.1.37 Utility Patterns (defer + static_let)

This tutorial covers two powerful utility macros: `defer` for Go-style scope-exit cleanup, and `static_let` for C-style persistent local variables.

```
require daslib/defer
require daslib/static_let
```

defer — scope-exit cleanup

`defer` moves a block of code to the `finally` section of the enclosing scope. It runs when the scope exits — whether normally or via an early return. This is Go's `defer` pattern: set up a resource, then immediately defer its cleanup, keeping the two related operations adjacent.

```
print("step 1\n")
defer() {
    print("cleanup (deferred)\n")
}
print("step 2\n")
print("step 3\n")
// output:
// step 1
// step 2
// step 3
// cleanup (deferred)
```

LIFO ordering

When multiple defers exist in the same scope, they run in LIFO order — the last deferred block runs first, just like Go:

```
defer() {
    print("first defer (runs last)\n")
}
defer() {
    print("second defer (runs first)\n")
}
print("main body\n")
// output:
// main body
```

(continues on next page)

(continued from previous page)

```
// second defer (runs first)
// first defer (runs last)
```

Early return

Deferred blocks run even when the function returns early. This makes `defer` ideal for resource cleanup:

```
def defer_early_return(do_early : bool) {
    defer() {
        print("cleanup always runs\n")
    }
    if (do_early) {
        print("returning early\n")
        return
    }
    print("reached the end\n")
}
```

Scope attachment

`defer` attaches to the nearest enclosing scope — so inside an `if` block, it only runs when that `if` block exits, not the function scope:

```
print("before if\n")
if (true) {
    defer() {
        print("deferred inside if\n")
    }
    print("inside if\n")
}
print("after if\n")
// output:
// before if
// inside if
// deferred inside if
// after if
```

Practical: paired acquire/release

The classic use case: acquire a resource, immediately defer its release:

```
acquire_resource("database")
defer() {
    release_resource("database")
}
acquire_resource("file")
defer() {
    release_resource("file")
```

(continues on next page)

(continued from previous page)

```
}
// On scope exit: file released first (LIFO), then database
```

Note: `defer` is transformed at compile time — the deferred block ALWAYS moves to `finally`, regardless of whether execution "reached" the `defer()` call at runtime. This differs from Go, where `defer` is runtime-conditional.

static_let — persistent locals

`static_let` makes local variable declarations persistent across calls. The variable is initialized once and retains its value, just like C's `static` local variables:

```
def call_counter() : int {
    static_let() {
        var count = 0
    }
    count ++
    return count
}

// call_counter() returns 1, 2, 3, ... on successive calls
```

Variables declared inside `static_let` are promoted to global scope but remain accessible by name in the declaring function.

One-time initialization

Use `static_let` for expensive one-time setup — the initializer runs once:

```
def get_lookup_table() : int {
    static_let() {
        var lookup <- [10, 20, 30, 40, 50]
    }
    var total = 0
    for (v in lookup) {
        total += v
    }
    return total
}
// The array is created once, reused on every call
```

Named static_let

The name argument helps disambiguate when you need multiple `static_let` blocks in different functions that might otherwise collide:

```
def named_counter_a() : int {
    static_let("counter_a") {
        var n = 0
    }
    n ++
    return n
}
```

Combining defer and static_let

A practical pattern: cache a result with `static_let`, and use `defer` to log when the function exits:

```
def cached_computation(input : int) : int {
    static_let() {
        var last_input = -1
        var last_result = 0
    }
    defer() {
        print("  exiting cached_computation({input})\n")
    }
    if (input == last_input) {
        print("  cache hit for {input}\n")
        return last_result
    }
    last_result = input * input + input
    last_input = input
    print("  computed {input} -> {last_result}\n")
    return last_result
}
```

Summary

Feature	Description
`defer`	Moves block to scope's `finally` section (LIFO)
`static_let`	Promotes local declarations to persistent globals
`static_let("name")`	Same, with a name prefix to avoid collisions
`static_let_finalize`	Same as `static_let`, but deletes on shutdown

See also:

Full source: `tutorials/language/37_utility_patterns.das`

Previous tutorial: *Pointers*

Next tutorial: *Random Numbers*

5.1.38 Random Numbers

This tutorial covers the `daslib/random` module — a deterministic pseudo-random number generator built on a linear congruential generator (LCG). All functions take a mutable `int4` seed state, making sequences reproducible when given the same starting seed.

```
require daslib/random
require math
```

Seeding the RNG

Every RNG function takes a `var int4&` seed. Create one from a single integer with `random_seed`. The same seed always produces the same sequence — useful for reproducible simulations:

```
var seed = random_seed(42)

// Same seed -> same sequence
var s1 = random_seed(42)
var s2 = random_seed(42)
print("{random_int(s1)} == {random_int(s2)}\n")  // true
```

Integer generation

```
var seed = random_seed(1)
print("{random_int(seed)}\n")       // 0..32767
print("{random_big_int(seed)}\n")   // 0..1073741823
print("{random_uint(seed)}\n")      // 0..4294967295
```

- `random_int` — range `0..LCG_RAND_MAX` (32767)
- `random_big_int` — wider range `0..32768*32768-1`
- `random_uint` — full 32-bit unsigned range

Float generation

`random_float` returns a value in `[0.0, 1.0)`. Map to a custom range with simple arithmetic:

```
var seed = random_seed(7)
let f = random_float(seed)          // 0.0 .. 1.0

// Map to [10, 20)
let lo = 10.0
let hi = 20.0
let mapped = lo + random_float(seed) * (hi - lo)
```

Vector generation

```
var seed = random_seed(13)
let ri = random_int4(seed)                // each component 0..32767
let rf = random_float4(seed)              // each component 0..1
let uv = random_unit_vector(seed)         // normalized direction
let sp = random_in_unit_sphere(seed)      // point inside unit sphere
let dk = random_in_unit_disk(seed)        // point inside unit disk (z=0)
```

- `random_unit_vector` — uniformly distributed direction (length = 1)
- `random_in_unit_sphere` — uniformly distributed point inside the unit sphere
- `random_in_unit_disk` — uniformly distributed point inside the unit disk (z component is always 0)

Infinite iterator

`each_random_uint` produces an infinite stream of random `uint` values. Use `break` or `take` to limit it:

```
var count = 0
for (val in each_random_uint(42)) {
    print("{val} ")
    count ++
    if (count >= 5) {
        break
    }
}
```

Practical examples

Dice roll

```
def roll_dice(var seed : int4&) : int {
    return (random_int(seed) % 6) + 1
}
```

Random pick from array

```
def random_pick(arr : array<string>; var seed : int4&) : string {
    let idx = random_int(seed) % length(arr)
    return arr[idx]
}
```

Fisher-Yates shuffle

```
def shuffle(var arr : array<int>; var seed : int4&) {
    var i = length(arr) - 1
    while (i > 0) {
        let j = random_int(seed) % (i + 1)
        let tmp = arr[i]
        arr[i] = arr[j]
        arr[j] = tmp
        i --
    }
}
```

Summary

Function	Returns
`random_seed(int)`	`int4` seed state
`random_int(seed)`	`int` 0..32767
`random_big_int(seed)`	`int` 0..1073741823
`random_uint(seed)`	`uint` full range
`random_float(seed)`	`float` 0..1
`random_int4(seed)`	`int4` each 0..32767
`random_float4(seed)`	`float4` each 0..1
`random_unit_vector(seed)`	`float3` length=1
`random_in_unit_sphere(s)`	`float3` inside unit sphere
`random_in_unit_disk(s)`	`float3` inside unit disk (z=0)
`each_random_uint(seed)`	infinite `iterator<uint>`

See also:

Full source: `tutorials/language/38_random.das`

Previous tutorial: *Utility Patterns (defer + static_let)*

Next tutorial: *Dynamic Type Checking*

5.1.39 Dynamic Type Checking

This tutorial covers runtime type checking and safe downcasting for class hierarchies using the `daslib/dynamic_cast_rtti` module. This requires `options rtti` to enable runtime type information.

```
options rtti
require daslib/dynamic_cast_rtti
```

Class hierarchy setup

All examples use a small shape hierarchy:

```
class Shape {
    name : string
    def abstract const area() : float
}

class Circle : Shape {
    radius : float
    def override const area() : float {
        return 3.14159265 * radius * radius
    }
}

class Rectangle : Shape {
    width : float
    height : float
    def override const area() : float {
        return width * height
    }
}

class Square : Rectangle {
    def Square(s : float) {
        width = s
        height = s
    }
}
```

Note the use of `def abstract const` and `def override const` — the `const` modifier makes `self` immutable, allowing the method to be called through const pointers.

is_instance_of

`is_instance_of` checks whether a class pointer is an instance of a given type at runtime, walking the RTTI chain. Works with both exact and derived types:

```
var c = new Circle(name = "circle", radius = 5.0)
let s_c : Shape? = c

print("{is_instance_of(s_c, type<Circle>)}\n")       // true
print("{is_instance_of(s_c, type<Rectangle>)}\n")     // false

// Derived types match base checks
var sq = new Square(name = "square")
let s_sq : Shape? = sq
print("{is_instance_of(s_sq, type<Rectangle>)}\n")    // true
print("{is_instance_of(s_sq, type<Shape>)}\n")        // true

// Null is never an instance of anything
```

(continues on next page)

(continued from previous page)

```
var np : Shape? = null
print("{is_instance_of(np, type<Shape>)}\n")          // false
```

dynamic_type_cast — safe downcast

`dynamic_type_cast` returns a typed pointer if the cast succeeds, or `null` if it fails. This is the "safe" downcast:

```
def describe_shape(s : Shape?) {
    let circle = dynamic_type_cast(s, type<Circle>)
    if (circle != null) {
        print("Circle: radius={circle.radius}, area={s->area()}\n")
        return
    }
    let rect = dynamic_type_cast(s, type<Rectangle>)
    if (rect != null) {
        print("Rectangle: {rect.width}x{rect.height}\n")
        return
    }
    print("Unknown shape: area={s->area()}\n")
}
```

force_dynamic_type_cast

`force_dynamic_type_cast` panics if the cast fails. Use when you are certain of the type and want a clear error if your assumption is wrong:

```
var c = new Circle(name = "c", radius = 7.0)
let s : Shape? = c
let fc = force_dynamic_type_cast(s, type<Circle>)
print("radius={fc.radius}\n")    // 7

// This would panic:
// let fr = force_dynamic_type_cast(s, type<Rectangle>)
```

is / as / ?as syntax sugar

After requiring `daslib/dynamic_cast_rtti`, you get syntactic sugar that maps to the functions above:

Syntax	Equivalent
`ptr is ClassName`	`is_instance_of(ptr, type<ClassName>)`
`ptr as ClassName`	`force_dynamic_type_cast(ptr, type<ClassName>)`
`ptr ?as ClassName`	`dynamic_type_cast(ptr, type<ClassName>)`

```
let s_c : Shape? = c

// is - type check
print("{s_c is Circle}\n")          // true
print("{s_c is Rectangle}\n")       // false
```

(continues on next page)

(continued from previous page)

```
// as - force cast (panics on failure)
let circle = s_c as Circle
print("radius={circle.radius}\n")

// ?as - safe cast (null on failure)
let maybe_rect = s_c ?as Rectangle
if (maybe_rect == null) {
    print("not a rectangle\n")
}
```

Practical: polymorphic processing

Combine `is` and `as` to process a heterogeneous collection of shapes:

```
var shapes : array<Shape?>
shapes |> push(new Circle(name = "sun", radius = 10.0))
shapes |> push(new Rectangle(name = "wall", width = 8.0, height = 3.0))

var total_area = 0.0
for (s in shapes) {
    if (s is Circle) {
        let c = s as Circle
        print("{c.name}: circle r={c.radius}\n")
    } elif (s is Square) {
        let sq = s as Square
        print("{sq.name}: square side={sq.width}\n")
    } elif (s is Rectangle) {
        let r = s as Rectangle
        print("{r.name}: rect {r.width}x{r.height}\n")
    }
    total_area += s->area()
}
print("total area: {total_area}\n")
```

Note: Check more-derived types first — `Square` must come before `Rectangle` since `Square` extends `Rectangle`.

Summary

Function / Syntax	Description
`is_instance_of(ptr, type<T>)`	`true` if ptr is instance of T via RTTI
`dynamic_type_cast(ptr, T)`	Returns `T?` or `null` on failure
`force_dynamic_type_cast(ptr, T)`	Returns `T?` or panics on failure
`ptr is ClassName`	Syntactic sugar for `is_instance_of`
`ptr as ClassName`	Syntactic sugar for force cast
`ptr ?as ClassName`	Syntactic sugar for safe cast

See also:

Classes — class inheritance and virtual methods.

Full source: `tutorials/language/39_dynamic_type_checking.das`

Previous tutorial: *Random Numbers*

Next tutorial: *Coroutines*

5.1.40 Coroutines

This tutorial covers generator-based coroutines using the `daslib/coroutines` module. Coroutines are functions that can suspend and resume, enabling cooperative multitasking without threads.

Prerequisites: *Iterators and Generators* (Tutorial 15).

```
require daslib/coroutines
```

Basic coroutines

A coroutine is a function annotated with `[coroutine]`. Under the hood it becomes a `generator<bool>` state machine. Use `co_continue()` to yield control (signal "still running"):

```
[coroutine]
def counting_coroutine() {
    print("count: 1\n")
    co_continue()
    print("count: 2\n")
    co_continue()
    print("count: 3\n")
    // Falls through - coroutine finishes
}
```

`cr_run` drives the coroutine to completion (consumes all yields):

```
var c <- counting_coroutine()
cr_run(c)
// output:
// count: 1
// count: 2
// count: 3
```

Manual stepping

Since a coroutine is just an `iterator<bool>`, you can step through it manually with a `for` loop:

```
[coroutine]
def step_coroutine() {
    print("step A\n")
    co_continue()
    print("step B\n")
    co_continue()
```

(continues on next page)

(continued from previous page)

```
    print("step C\n")
}

var c <- step_coroutine()
for (running in c) {
    print("-- yielded (running={running}) --\n")
}
print("-- coroutine done --\n")
```

Each iteration of the `for` loop advances the coroutine to the next `co_continue()` or end.

Coroutines with state

Coroutines can have local variables that persist across yields. Each `co_continue()` suspends, and on resume the locals are intact:

```
[coroutine]
def countdown(n : int) {
    var i = n
    while (i > 0) {
        print("{i}...\n")
        i --
        if (i > 0) {
            co_continue()
        }
    }
}

var c <- countdown(4)
cr_run(c)
print("liftoff!\n")
// output: 4... 3... 2... 1... liftoff!
```

cr_run_all — cooperative scheduling

`cr_run_all` takes an array of `Coroutine` (`iterator<bool>`) and drives them round-robin until all are done:

```
[coroutine]
def worker(name : string; steps : int) {
    for (i in range(steps)) {
        print("{name}: step {i + 1}/{steps}\n")
        if (i < steps - 1) {
            co_continue()
        }
    }
}

var tasks : Coroutines
tasks |> emplace <| worker("alpha", 3)
tasks |> emplace <| worker("beta", 2)
```

(continues on next page)

(continued from previous page)

```
tasks |> emplace <| worker("gamma", 4)
cr_run_all(tasks)
```

Each coroutine gets one step per round. When a coroutine finishes, it is removed from the array.

co_await — waiting for a sub-coroutine

`co_await` suspends the current coroutine until a sub-coroutine finishes. This is useful for composing coroutines hierarchically:

```
[coroutine]
def load_data() {
    print("loading data... (tick 1)\n")
    co_continue()
    print("loading data... (tick 2)\n")
    co_continue()
    print("data loaded!\n")
}

[coroutine]
def process_pipeline() {
    print("pipeline: start\n")
    co_continue()
    print("pipeline: awaiting load_data\n")
    co_await <| load_data()
    print("pipeline: processing loaded data\n")
    co_continue()
    print("pipeline: done\n")
}
```

yeild_from — delegating to a sub-generator

`yeild_from` yields all values from a sub-iterator. It works with any generator, not just coroutines:

```
def numbers_gen() : iterator<int> {
    return <- generator<int>() <| $() {
        yield 10
        yield 20
        yield 30
        return false
    }
}

def combined_gen() : iterator<int> {
    return <- generator<int>() <| $() {
        yield 1
        yeild_from <| numbers_gen()
        yield 2
        return false
    }
```

(continues on next page)

(continued from previous page)

```
}

for (v in combined_gen()) {
    print("{v} ")
}
// output: 1 10 20 30 2
```

Practical: cooperative batch processing

Simulate processing items in batches, with multiple workers cooperatively sharing time:

```
[coroutine]
def batch_processor(name : string; total_items : int) {
    let batch_size = 2
    var pos = 0
    while (pos < total_items) {
        var end = min(pos + batch_size, total_items)
        print("{name}: batch [{pos}..{end - 1}]\n")
        pos = end
        if (pos < total_items) {
            co_continue()
        }
    }
}

var tasks : Coroutines
tasks |> emplace <| batch_processor("worker-A", 5)
tasks |> emplace <| batch_processor("worker-B", 3)
cr_run_all(tasks)
```

Summary

Feature	Description
`[coroutine]`	Annotation that transforms function to generator
`co_continue()`	Yield control (still running)
`cr_run(c)`	Drive one coroutine to completion
`cr_run_all(tasks)`	Drive array of coroutines round-robin
`co_await <| sub()`	Suspend until sub-coroutine finishes
`yeild_from <| gen()`	Delegate to sub-generator
`Coroutines`	Type alias for `array<iterator<bool>>`

See also:

Iterators and Generators — generator language reference.

Full source: `tutorials/language/40_coroutines.das`

Previous tutorial: *Dynamic Type Checking*

Next tutorial: *Serialization (archive)*

5.1.41 Serialization (archive)

This tutorial covers binary serialization using the `daslib/archive` module. The module provides automatic serialization for all built-in types, structs, arrays, tables, tuples, and variants.

```
require daslib/archive
```

Primitive serialization

`mem_archive_save` serializes any value to an `array<uint8>`. `mem_archive_load` deserializes it back:

```
var x = 42
var data <- mem_archive_save(x)
print("int data size: {length(data)} bytes\n")    // 4 bytes

var loaded_x : int
mem_archive_load(data, loaded_x)
print("loaded int: {loaded_x}\n")                 // 42
```

This works for `int`, `float`, `string`, and all other primitive types.

Struct serialization

Structs are serialized field by field automatically:

```
struct Player {
    name : string
    hp : int
    x : float
    y : float
}

var player = Player(name = "Hero", hp = 100, x = 1.5, y = 2.5)
var data <- mem_archive_save(player)

var loaded : Player
mem_archive_load(data, loaded)
print("{loaded.name}, hp={loaded.hp}\n")    // Hero, hp=100
```

Array serialization

Dynamic arrays serialize their length followed by all elements:

```
var scores <- [10, 20, 30, 40, 50]
var data <- mem_archive_save(scores)

var loaded : array<int>
mem_archive_load(data, loaded)
// loaded contains [10, 20, 30, 40, 50]
```

Table serialization

Tables serialize their length followed by key-value pairs:

```
var inventory : table<string; int>
inventory |> insert("sword", 1)
inventory |> insert("potion", 5)
inventory |> insert("arrow", 20)

var data <- mem_archive_save(inventory)
var loaded : table<string; int>
mem_archive_load(data, loaded)
```

Note: Table iteration order may vary, but all key-value pairs are preserved.

Tuple serialization

```
var pair : tuple<name:string; score:int>
pair.name = "Alice"
pair.score = 99

var data <- mem_archive_save(pair)
var loaded : tuple<name:string; score:int>
mem_archive_load(data, loaded)
print("name={loaded.name}, score={loaded.score}\n")
```

Variant serialization

Variants serialize the active variant index plus its value. Requires `unsafe` for variant field access:

```
variant Value {
    i : int
    f : float
    s : string
}

unsafe {
    var v1 = Value(i = 42)
    var data <- mem_archive_save(v1)
    var loaded : Value
    mem_archive_load(data, loaded)
    if (loaded is i) {
        print("loaded: int = {loaded.i}\n")    // 42
    }
}
```

Nested structures

Serialization is recursive — structs containing arrays, tables, and other structs are handled automatically:

```
struct Inventory {
    items : array<string>
    counts : table<string; int>
}

struct GameState {
    player : Player
    inventory : Inventory
    level : int
}

var state : GameState
state.player = Player(name = "Knight", hp = 80, x = 10.0, y = 20.0)
state.level = 3
state.inventory.items |> push("sword")
state.inventory.counts |> insert("sword", 1)

var data <- mem_archive_save(state)
var loaded : GameState
mem_archive_load(data, loaded)
```

Manual Archive usage

For full control, create an `Archive` and `MemSerializer` manually. This lets you write multiple values into a single byte stream:

```
// Writing phase
var writer = new MemSerializer()
var warch = Archive(reading = false, stream = writer)
var name = "save_001"
var score = 9999
var tags <- ["rpg", "fantasy"]
warch |> serialize(name)
warch |> serialize(score)
warch |> serialize(tags)
var data <- writer->extractData()

// Reading phase
var reader = new MemSerializer(data)
var rarch = Archive(reading = true, stream = reader)
var r_name : string
var r_score : int
var r_tags : array<string>
rarch |> serialize(r_name)
rarch |> serialize(r_score)
rarch |> serialize(r_tags)
```

The same `serialize` function works for both reading and writing — the `Archive.reading` flag determines the direction.

Custom serialize

You can override the serialization of any type by defining a `serialize` overload with the exact signature `def serialize(var arch : Archive; var val : YourType&)`. This overload is more specific than the generic struct serializer, so it wins — as long as the `serialize` call happens within the module that defines the overload.

Note: `mem_archive_save` / `mem_archive_load` resolve `serialize` inside the archive module, so they won't pick up overloads defined in user code. Use the manual `Archive` + `MemSerializer` pattern instead.

Here a `Color` struct stores channels as floats but serializes as 3 compact bytes:

```
struct Color {
    r : float = 0.0
    g : float = 0.0
    b : float = 0.0
}

def serialize(var arch : Archive; var c : Color&) {
    if (arch.reading) {
        var rb, gb, bb : uint8
        arch |> serialize_raw(rb)
        arch |> serialize_raw(gb)
        arch |> serialize_raw(bb)
        c.r = float(rb) / 255.0
        c.g = float(gb) / 255.0
        c.b = float(bb) / 255.0
    } else {
        var rb = uint8(clamp(c.r * 255.0, 0.0, 255.0))
        var gb = uint8(clamp(c.g * 255.0, 0.0, 255.0))
        var bb = uint8(clamp(c.b * 255.0, 0.0, 255.0))
        arch |> serialize_raw(rb)
        arch |> serialize_raw(gb)
        arch |> serialize_raw(bb)
    }
}
```

Using it with the manual archive pattern:

```
var c = Color(r = 1.0, g = 0.5, b = 0.0)
var writer = new MemSerializer()
var warch = Archive(reading = false, stream = writer)
warch |> serialize(c)
var data <- writer->extractData()
print("size: {length(data)} bytes\n")     // 3 (not 12)

var reader = new MemSerializer(data)
var rarch = Archive(reading = true, stream = reader)
var loaded = Color()
rarch |> serialize(loaded)
```

The custom overload is also picked up by the generic array serializer when the call-site is in the same module:

```
var colors <- [Color(r=1.0, g=0.0, b=0.0), Color(r=0.0, g=1.0, b=0.0)]
var writer = new MemSerializer()
var warch = Archive(reading = false, stream = writer)
warch |> serialize(colors)      // 4 + 2×3 = 10 bytes
```

Summary

Function	Description
`mem_archive_save(value)`	Serialize any value to `array<uint8>`
`mem_archive_load(data, value)`	Deserialize from `array<uint8>` into value
`serialize(archive, value)`	Read or write depending on `archive.reading`
`serialize_raw(archive, value)`	Raw byte read/write (no type dispatch)
`Archive`	Struct combining stream + direction
`MemSerializer`	In-memory byte stream (reading or writing)

See also:

Full source: `tutorials/language/41_serialization.das`

Previous tutorial: *Coroutines*

Next tutorial: *Testing Tools (faker + fuzzer)*

5.1.42 Testing Tools (faker + fuzzer)

This tutorial covers two testing modules: `daslib/faker` for generating random test data, and `daslib/fuzzer` for running stress tests that detect crashes and invariant violations.

Prerequisites: *Testing with dastest* (Tutorial 27), *Random Numbers* (Tutorial 38).

```
require daslib/faker
require daslib/fuzzer
```

Faker basics

A `Faker` struct produces random values for every built-in type. It wraps an infinite random `uint` iterator internally:

```
var fake <- Faker()

print("{fake |> random_int}\n")      // random int (full range)
print("{fake |> random_uint}\n")     // random uint (full range)
print("{fake |> random_float}\n")    // random float (raw bits)
print("{fake |> random_double}\n")   // random double (raw bits)

// Random bool (derived from random_uint)
let b = (fake |> random_uint) % 2u == 0u

delete fake
```

Note: `random_float` and `random_double` generate raw bit patterns — the values can be NaN, Inf, or denormalized. This is intentional for fuzz testing.

Random strings and file names

```
var fake <- Faker()
let s = fake |> any_string()          // random characters
let fn = fake |> any_file_name        // alphanumeric + underscore + dot
let ls = fake |> long_string()        // up to max_long_string bytes
delete fake
```

- `any_string` — random characters up to the regex generator limit
- `any_file_name` — file-name-safe characters
- `long_string` — may be very long (configurable via `fake.max_long_string`)

Random vectors

Faker generates vectors of all sizes for every numeric type:

```
var fake <- Faker()
print("{fake |> random_int2}\n")
print("{fake |> random_int3}\n")
print("{fake |> random_float2}\n")
print("{fake |> random_float3}\n")
delete fake
```

Random dates

Faker generates random dates within a configurable year range:

```
var fake <- Faker()
print("{fake |> date}\n")     // "DayOfWeek, MonthName DD, YYYY"
print("{fake |> day}\n")      // "DayOfWeek"
delete fake
```

Customizing Faker

Faker has configurable fields to control the output:

```
var fake <- Faker()
fake.min_year = 2020u          // restrict year range
fake.total_years = 5u          // 2020-2025
fake.max_long_string = 32u     // limit long_string length
delete fake
```

fuzz — silent crash detection

`fuzz(count, block)` runs a block `count` times, catching any panics. Use it to test that your code handles arbitrary input gracefully:

```
def safe_divide(a, b : int) : int {
    if (b == 0) {
        return 0
    }
    return a / b
}

var fake <- Faker()
fuzz(100) {
    let a = fake |> random_int
    let b = fake |> random_int
    safe_divide(a, b)
}
delete fake
```

If any iteration panics, `fuzz` catches it silently and continues.

fuzz_debug — verbose crash detection

`fuzz_debug` is identical to `fuzz` but does NOT catch panics. Replace `fuzz` with `fuzz_debug` when you need to see the actual error message and stack trace:

```
var fake <- Faker()
fuzz_debug(50) {
    let a = fake |> random_int
    let b = fake |> random_int
    safe_divide(a, b)
}
delete fake
```

Property-based testing

Combine faker + fuzzer for property-based testing: generate random inputs and verify that invariants always hold:

```
def clamp_value(x, lo, hi : int) : int {
    if (x < lo) {
        return lo
    }
    if (x > hi) {
        return hi
    }
    return x
}

var fake <- Faker()
var violations = 0
fuzz(1000) {
```

(continues on next page)

(continued from previous page)

```
    let x = fake |> random_int
    let result = clamp_value(x, -100, 100)
    if (result < -100 || result > 100) {
        violations ++
    }
}
print("violations: {violations}/1000\n")    // 0
delete fake
```

Testing string operations

Faker's `any_string` is useful for testing string-processing functions:

```
var fake <- Faker()
var failures = 0
fuzz(100) {
    let s = fake |> any_string()
    let reversed = reverse_string(s)
    let double_rev = reverse_string(reversed)
    if (double_rev != s) {
        failures ++
    }
}
delete fake
```

Summary

Function	Description
`Faker()`	Create faker with default RNG
`fake |> random_int`	Random int (full range)
`fake |> random_uint`	Random uint (full range)
`fake |> random_float`	Random float (raw bits — may be NaN/Inf)
`fake |> any_string()`	Random string
`fake |> any_file_name`	Random file-name-safe string
`fake |> long_string()`	Random byte string (up to `max_long_string`)
`fake |> date`	Random date string
`fake |> day`	Random day-of-week string
`fuzz(n) <| block`	Run block n times, catch panics
`fuzz_debug(n) <| block`	Run block n times, panics propagate

See also:

Testing — testing framework tutorial.

Random Numbers — random number generation.

Full source: `tutorials/language/42_testing_tools.das`

Previous tutorial: *Serialization (archive)*

Next tutorial: *Interfaces*

5.1.43 Interfaces

This tutorial covers **interface-based polymorphism** using the `daslib/interfaces` module. Interfaces let you define abstract method contracts that structs can implement, enabling polymorphic dispatch without class inheritance.

Prerequisites: *Dynamic Type Checking* (Tutorial 39).

```
require daslib/interfaces
```

Defining an interface

Use `[interface]` on a class to declare it as an interface. An interface may contain only function-typed fields — abstract methods or default implementations:

```
[interface]
class IGreeter {
    def abstract greet(name : string) : string
}
```

Implementing an interface

Use `[implements(IFoo)]` on a struct to generate the proxy class and getter. Implement each method with the `InterfaceName`method` naming convention:

```
[implements(IGreeter)]
class FriendlyGreeter {
    def FriendlyGreeter() {
        pass
    }
    def IGreeter`greet(name : string) : string {
        return "Hey, {name}!"
    }
}
```

`is` / `as` / `?as` operators

The `InterfaceAsIs` variant macro enables three operators that work with interface types:

```
var g = new FriendlyGreeter()

// is - compile-time check (zero runtime cost)
print("{g is IGreeter}\n")          // true

// as - returns the interface proxy
var iface = g as IGreeter
iface->greet("Alice")

// ?as - null-safe: returns null when the pointer is null
var nothing : FriendlyGreeter?
var safe = nothing ?as IGreeter     // null
```

`is` is resolved entirely at compile time — the result is baked into the program as a boolean constant.

Multiple interfaces

A struct can implement any number of interfaces by listing multiple `[implements(...)]` annotations:

```
[interface]
class IDrawable {
    def abstract draw(x, y : int) : void
}

[interface]
class ISerializable {
    def abstract serialize : string
}

[implements(IDrawable), implements(ISerializable)]
class Sprite {
    name : string
    def Sprite(n : string) { name = n }
    def IDrawable`draw(x, y : int) {
        print("Sprite \"{name}\" at ({x},{y})\n")
    }
    def ISerializable`serialize() : string {
        return "sprite:{name}"
    }
}
```

Polymorphic dispatch

Interfaces enable true polymorphic dispatch — pass interface proxies to functions that accept the interface type:

```
def draw_all(var objects : array<IDrawable?>) {
    for (obj in objects) {
        obj->draw(0, 0)
    }
}

var drawables : array<IDrawable?>
drawables |> push(circle as IDrawable)
drawables |> push(sprite as IDrawable)
draw_all(drawables)
```

Interface inheritance

Interfaces can extend other interfaces using normal class inheritance syntax. A struct that implements a derived interface automatically supports `is`/`as`/`?as` for all ancestor interfaces:

```
[interface]
class IAnimal {
    def abstract name : string
    def abstract sound : string
}
```

(continues on next page)

(continued from previous page)

```
[interface]
class IPet : IAnimal {
    def abstract owner : string
}

[implements(IPet)]
class Dog {
    dog_name : string
    owner_name : string
    def Dog(n, o : string) { dog_name = n; owner_name = o }
    def IPet`name() : string { return dog_name }
    def IPet`sound() : string { return "Woof" }
    def IPet`owner() : string { return owner_name }
}
```

Now `Dog` supports both `IPet` and `IAnimal`:

```
var d = new Dog("Rex", "Alice")
print("{d is IPet}\n")       // true
print("{d is IAnimal}\n")    // true

var animal = d as IAnimal
animal->name()               // "Rex"
animal->sound()              // "Woof"
```

Default method implementations

Non-abstract methods in an interface class provide default implementations. The implementing struct only needs to override the abstract methods — defaults are inherited by the proxy:

```
[interface]
class ILogger {
    def abstract log_name : string
    // Default - optional to override
    def format(message : string) : string {
        return "[{self->log_name()}] {message}"
    }
}

[implements(ILogger)]
class SimpleLogger {
    def SimpleLogger() { pass }
    def ILogger`log_name() : string { return "simple" }
    // format() uses ILogger's default
}
```

A struct that overrides the default provides its own implementation:

```
[implements(ILogger)]
class FancyLogger {
```

(continues on next page)

(continued from previous page)

```
    def FancyLogger() { pass }
    def ILogger`log_name() : string { return "fancy" }
    def ILogger`format(message : string) : string {
        return "*** [fancy] {message} ***"
    }
}
```

Completeness checking

If an implementing struct is missing an abstract method, the compiler reports an error at compile time:

```
error[30111]: Foo does not implement IBar.method
```

Methods with default implementations are optional — the proxy inherits the default from the interface class. Only abstract methods (`def abstract`) are required.

Quick reference

Syntax	Description
`[interface]`	Mark a class as an interface
`[implements(IFoo)]`	Generate proxy and getter for `IFoo`
`def abstract method(...) : T`	Declare an abstract method (required)
`def method(...) : T { ... }`	Declare a default method (optional to override)
`def IFoo`method(...)`	Implement (or override) a method on a struct
`ptr is IFoo`	Compile-time check (true/false)
`ptr as IFoo`	Get the interface proxy
`ptr ?as IFoo`	Null-safe proxy access
`class IChild : IParent`	Interface inheritance

See also:

Dynamic Type Checking — `is`/`as` on class hierarchies (Tutorial 39).

Variant Macros — how the `InterfaceAsIs` macro works internally (Macro Tutorial 8).

Interfaces module — standard library reference.

Full source: `tutorials/language/43_interfaces.das`

Previous tutorial: *Testing Tools (faker + fuzzer)*

Next tutorial: *Compiling and Running Programs at Runtime*

5.1.44 Compiling and Running Programs at Runtime

This tutorial covers **compiling and running daslang programs from daslang** — dynamically compiling source code at runtime, simulating it into a runnable context, and invoking exported functions across context boundaries.

Prerequisites: *Job Queue (jobque)* (Tutorial 35) for the channel section.

```
options gen2
options multiple_contexts   // required when holding smart_ptr<Context>

require rtti
require debugapi
require daslib/jobque_boost
```

Compile from string

The simplest way to compile daslang at runtime is `compile`, which takes a module name, source text, and `CodeOfPolicies`. The callback receives `(ok : bool, program : smart_ptr<Program>, issues : string)`.

Always set `cop.threadlock_context = true` — this is required for `invoke_in_context` to work:

```
using <| $(var cop : CodeOfPolicies) {
    cop.threadlock_context = true
    compile("inline", src, cop) <| $(ok, program, issues) {
        if (!ok) {
            print("compile error: {issues}\n")
            return
        }
        simulate(program) <| $(sok; context; serrors) {
            if (!sok) {
                print("simulate error: {serrors}\n")
                return
            }
            unsafe {
                invoke_in_context(context, "hello")
            }
        }
    }
}
// output:
//   hello from compiled code!
```

Compile from file

`compile_file` compiles a `.das` file from disk. It requires a `FileAccess` (for file I/O) and a `ModuleGroup` (for module resolution). Both must stay alive during compile + simulate:

```
var inscope access <- make_file_access("")
using <| $(var mg : ModuleGroup) {
    using <| $(var cop : CodeOfPolicies) {
        cop.threadlock_context = true
```

(continues on next page)

(continued from previous page)

```
        compile_file("tutorials/language/44_helper.das", access,
                     unsafe(addr(mg)), cop) <| $(ok, program, issues) {
            if (!ok) {
                print("compile error: {issues}\n")
                return
            }
            simulate(program) <| $(sok; context; serrors) {
                if (!sok) {
                    print("simulate error: {serrors}\n")
                    return
                }
                unsafe {
                    invoke_in_context(context, "main")
                }
            }
        }
    }
}
// output:
//   hello from 44_helper!
```

Virtual file system

You can compile code that does not exist on disk by injecting virtual files into the `FileAccess` object with `set_file_source`. This is useful for code generation, REPLs, and eval-like tools:

```
var inscope access <- make_file_access("")
access |> set_file_source("__generated.das", generated_code)

// Now compile_file("__generated.das", access, ...) will find it.
// Other modules on disk remain accessible through the same access.
```

Invoke with arguments

`invoke_in_context` can pass up to 10 arguments by value. It always returns `void` — see later sections for how to retrieve results.

Use `has_function` to check whether a function exists before calling it:

```
if (has_function(*context, "sum3")) {
    unsafe {
        invoke_in_context(context, "sum3", 10, 20, 30)
    }
}
// output:
//   sum3 = 60
```

Note: `has_function` takes a `Context` reference, so you must dereference the smart pointer with `*context`.

Reading results via global variables

Since `invoke_in_context` returns void, one way to get a result is to have the child store it in a global variable, then read it back with `get_context_global_variable`.

The pointer returned is into the child context's memory — copy the value immediately before the context goes out of scope:

```
// Call compute(7) - sets global `result` to 7*7+1 = 50
unsafe {
    invoke_in_context(context, "compute", 7)
}
// Read back the global variable "result"
let ptr = unsafe(get_context_global_variable(context, "result"))
if (ptr != null) {
    let value = *unsafe(reinterpret<int?> ptr)
    print("result = {value}\n")
}
// output:
//   result = 50
```

Passing results via pointer argument

Another pattern is to pass a pointer from the host into the child. The child writes through the pointer, and the host reads it after `invoke_in_context` returns:

```
// Child function: def store_via_ptr(val : int; var dst : int?)
var output = 0
unsafe {
    invoke_in_context(context, "store_via_ptr", 5, addr(output))
}
print("output = {output}\n")    // 5 * 10 = 50
```

Using channels for cross-context results

Channels (from `daslib/jobque_boost`) are cross-context safe — ideal for collecting structured results from child contexts. The host creates a channel with `with_channel(1)`, passes it as a `Channel?` argument to the child via `invoke_in_context`, and the child pushes results with `push_clone` + `notify`. The host drains the channel with `for_each_clone`. Data arrives as a temporary type (`TT#`) ensuring proper copy from the foreign heap.

Use `notify`, **not** `notify_and_release`. When a lambda captures a channel, its reference count is incremented, so `notify_and_release` releases that extra reference and nulls the variable. With `invoke_in_context` there is no lambda — the child does not own the channel and no extra reference was added — so plain `notify` is correct.

The child script needs `require daslib/jobque_boost` to use channel operations. Use `compile_file` with `make_file_access("")` so the child can resolve daslib modules from disk:

```
struct IntResult {
    value : int
}

// Child script receives Channel?, pushes result, notifies
let child_src = "..."   // defines produce(var ch : Channel?)
```

(continues on next page)

(continued from previous page)

```
var inscope access <- make_file_access("")
access |> set_file_source("__child.das", child_src)
// ... compile_file + simulate ...
with_channel(1) $(ch) {
    unsafe {
        invoke_in_context(context, "produce", ch)
    }
    ch |> for_each_clone() $(val : IntResult#) {
        print("channel received: {val.value}\n")
    }
}
```

Error handling

Compilation and simulation can fail. Always check the `ok` / `sok` flags. Runtime errors in the child context can be caught with `try`/`recover`:

```
// 1) Compilation error
compile("bad", bad_src, cop) <| $(ok, program, issues) {
    if (!ok) {
        print("compile error: {issues}\n")
        return
    }
}

// 2) Runtime error
try {
    unsafe {
        invoke_in_context(context, "crash")
    }
} recover {
    print("runtime error caught\n")
}
```

Quick reference

`compile(name, src, cop) <| ...`	Compile from source string
`compile_file(name, access, mg, cop)`	Compile from file via FileAccess + ModuleGroup
`simulate(program) <| ...`	Simulate a program into a Context
`invoke_in_context(ctx, name, ...)`	Call an [export] function (returns void)
`has_function(*ctx, name)`	Check if function exists in context
`get_context_global_variable(ctx, n)`	Read global variable pointer from context
`make_file_access("")`	Create disk-backed FileAccess
`set_file_source(access, name, src)`	Inject a virtual file
`with_channel(N) $(ch) { ... }`	Create channel; pass `ch` to child context
`push_clone` / `notify`	Child pushes results, notifies host
`for_each_clone`	Host drains channel results as `TT#`
`options multiple_contexts`	Required when holding smart_ptr<Context>
`cop.threadlock_context = true`	Required for invoke_in_context

See also:

Job Queue — channels, lock boxes, and threading (Tutorial 35).

Contexts — language reference for context semantics.

Full source: `tutorials/language/44_compile_and_run.das`

Helper file: `tutorials/language/44_helper.das`

Previous tutorial: *Interfaces*

Next tutorial: *Debug Agents*

5.1.45 Debug Agents

This tutorial covers **debug agents** — persistent objects that live in their own separate context and can intercept runtime events, collect state, and host shared data accessible from any context.

Prerequisites: *Compiling and Running Programs at Runtime* (Tutorial 44) for `invoke_in_context` basics.

```
options gen2

require debugapi
require rtti
```

Creating and installing a debug agent

A debug agent is a class that extends `DapiDebugAgent`. The fundamental pattern is:

1. Define a class that extends `DapiDebugAgent`
2. Write a setup function that creates the agent and installs it under a name with `install_new_debug_agent`
3. Call `fork_debug_agent_context(@@setup)` to clone the current context and run the setup function in it

The agent lives in its own "agent context" — a separate copy of the program that stays resident:

```
class CounterAgent : DapiDebugAgent {
    count : int = 0
}

def install_counter(ctx : Context) {
    install_new_debug_agent(new CounterAgent(), "counter")
}

def demo_create_agent() {
    print("  has 'counter' = {has_debug_agent_context("counter")}\n")
    fork_debug_agent_context(@@install_counter)
    print("  has 'counter' = {has_debug_agent_context("counter")}\n")
// output:
//   has 'counter' = false
//   has 'counter' = true
```

Intercepting log output with onLog

`DapiDebugAgent` has an `onLog` method called whenever any context prints or logs. If `onLog` returns `true`, the default output to stdout is suppressed. If it returns `false`, output proceeds normally.

This is how profiling tools, IDE log panels, and custom loggers intercept program output:

```
var log_intercept_count : int = 0

class LogAgent : DapiDebugAgent {
    def override onLog(context : Context?; at : LineInfo const?;
                       level : int; text : string#) : bool {
        log_intercept_count++
        return false  // don't suppress - let output reach stdout
    }
}

def install_log_agent(ctx : Context) {
    install_new_debug_agent(new LogAgent(), "log_watcher")
}

[export, pinvoke]
def read_log_count(var result : int?) {
    unsafe {
        *result = log_intercept_count
    }
}

// After forking + installing, each print/to_log triggers onLog
fork_debug_agent_context(@@install_log_agent)
print("  hello through agent\n")
to_log(LOG_INFO, "  info message\n")

var count = 0
unsafe {
    invoke_in_context(get_debug_agent_context("log_watcher"),
        "read_log_count", addr(count))
}
print("  log_intercept_count >= 2: {count >= 2}\n")
// output:
//   hello through agent
//   info message
//   log_intercept_count >= 2: true
```

Calling functions in the agent context

Use `invoke_in_context` to call `[export, pinvoke]` functions in the agent context. `get_debug_agent_context(name)` returns the agent's `Context`. Functions run in that context, so they see the agent's copy of module-level variables — not the caller's.

The `[pinvoke]` annotation is required — it enables the context mutex needed for cross-context invocation.

To return values, pass a pointer to a result variable:

```
var agent_counter : int = 0

[export, pinvoke]
def agent_increment() {
    agent_counter++
}

[export, pinvoke]
def agent_get(var result : int?) {
    unsafe {
        *result = agent_counter
    }
}

def demo_invoke_in_context() {
    unsafe {
        invoke_in_context(get_debug_agent_context("counter"), "agent_increment")
        invoke_in_context(get_debug_agent_context("counter"), "agent_increment")
        invoke_in_context(get_debug_agent_context("counter"), "agent_increment")
    }
    var result = 0
    unsafe {
        invoke_in_context(get_debug_agent_context("counter"), "agent_get", addr(result))
    }
    print("  agent_counter (in agent) = {result}\n")
    print("  agent_counter (local)    = {agent_counter}\n")
// output:
//   agent_counter (in agent) = 3
//   agent_counter (local)    = 0
```

Calling agent methods with invoke_debug_agent_method

`invoke_debug_agent_method` calls a method on the agent's class instance directly — no `[export, pinvoke]` helper functions needed. The agent's `self` is passed automatically.

Syntax: `invoke_debug_agent_method("agent_name", "method", args...)`

```
class CalcAgent : DapiDebugAgent {
    accumulator : int = 0
    def add(amount : int) {
        self.accumulator += amount
    }
    def get_result(var result : int?) {
        unsafe {
```

(continues on next page)

(continued from previous page)

```
            *result = self.accumulator
        }
    }
}

def install_calc_agent(ctx : Context) {
    install_new_debug_agent(new CalcAgent(), "calc")
}

fork_debug_agent_context(@@install_calc_agent)

unsafe {
    invoke_debug_agent_method("calc", "add", 10)
    invoke_debug_agent_method("calc", "add", 20)
    invoke_debug_agent_method("calc", "add", 12)
}

var result = 0
unsafe {
    invoke_debug_agent_method("calc", "get_result", addr(result))
}
print("  accumulator = {result}\n")
// output:
//   accumulator = 42
```

State collection — onCollect and onVariable

`onCollect` is called when `collect_debug_agent_state` is triggered. The agent can report custom variables via `report_context_state`. `onVariable` receives each reported variable — this is how IDE debuggers show custom watch variables and application diagnostics:

```
class StateAgent : DapiDebugAgent {
    collection_count : int = 0
    def override onCollect(var ctx : Context; at : LineInfo) : void {
        collection_count++
        unsafe {
            let tinfo = typeinfo rtti_typeinfo(collection_count)
            report_context_state(ctx, "Diagnostics", "collection_count",
                unsafe(addr(tinfo)), unsafe(addr(collection_count)))
        }
    def override onVariable(var ctx : Context; category, name : string;
                            info : TypeInfo; data : void?) : void {
        unsafe {
            let value = sprint_data(data, addr(info), print_flags.singleLine)
            print("  {category}: {name} = {value}\n")
        }
}

// Trigger collection
collect_debug_agent_state(this_context(), get_line_info(1))
// output:
```

(continues on next page)

(continued from previous page)

```
//   Diagnostics: collection_count = 1
```

Agent existence checks

`has_debug_agent_context(name)` checks if a named agent exists. Always check before accessing the context to avoid panics:

```
print("  has 'counter'  = {has_debug_agent_context("counter")}\n")
print("  has 'missing'  = {has_debug_agent_context("missing")}\n")
// output:
//   has 'counter'  = true
//   has 'missing'  = false
```

Auto-start module pattern

In modules, agents are installed automatically via a `[_macro]` function. Four guards ensure safe, single installation:

```
[_macro]
def private auto_start() {
    if (is_compiling_macros_in_module("my_module") && !is_in_completion()) {
        if (!is_in_debug_agent_creation()) {
            if (!has_debug_agent_context("my_agent")) {
                fork_debug_agent_context(@@my_agent_setup)
            }
        }
    }
}
```

The guards prevent:

- Running outside the module's own compilation
- Running during IDE code completion
- Recursive agent creation
- Duplicate installation

Plain agent as named context host

A common pattern is to create a plain `DapiDebugAgent` (no overrides) just to own a named context. Module-level variables in that context become shared state accessible via `invoke_in_context`. This is the foundation of the `[apply_in_context]` pattern (Tutorial 46):

```
var shared_data : int = 0

[export, pinvoke]
def add_data(amount : int) {
    shared_data += amount
}
```

(continues on next page)

(continued from previous page)

```
[export, pinvoke]
def get_data(var result : int?) {
    unsafe {
        *result = shared_data
    }
}

def install_data_host(ctx : Context) {
    install_new_debug_agent(new DapiDebugAgent(), "data_host")
}

// Multiple calls accumulate in the agent's copy
unsafe {
    invoke_in_context(get_debug_agent_context("data_host"), "add_data", 10)
    invoke_in_context(get_debug_agent_context("data_host"), "add_data", 20)
}
// output:
//   shared_data (in agent) = 30
//   shared_data (local)    = 0
```

Shutting down a debug agent

`delete_debug_agent_context` removes an agent by name. It notifies all other agents via `onUninstall`, then safely destroys the agent and its context:

```
// Remove the agent
delete_debug_agent_context("data_host")

print("has 'data_host' = {has_debug_agent_context("data_host")}\n")
// output: has 'data_host' = false

// Deleting a non-existent agent is a safe no-op
delete_debug_agent_context("data_host")
```

Use this when a profiling session ends, a debug tool is closed, or during test teardown to ensure agents do not leak across test files.

Quick reference

`fork_debug_agent_context(@@fn)`	Clone context and call setup function in clone
`install_new_debug_agent(agent, name)`	Register agent under a global name
`has_debug_agent_context(name)`	Check if named agent exists
`get_debug_agent_context(name)`	Get agent's Context for invoke_in_context
`delete_debug_agent_context(name)`	Remove agent by name (safe no-op if missing)
`invoke_in_context(ctx, fn_name, ...)`	Call [export, pinvoke] function in agent context
`invoke_debug_agent_method(name, meth, ...)`	Call a method on the agent's class instance
`collect_debug_agent_state(ctx, line)`	Trigger onCollect on all agents
`report_context_state(ctx, cat, name, ...)`	Report variable from onCollect to onVariable
`is_in_debug_agent_creation()`	True during fork_debug_agent_context
`to_log(level, text)`	Log message — routed through onLog if agents exist
`[pinvoke]`	Annotation enabling context mutex

See also:

Full source: `tutorials/language/45_debug_agents.das`

Cross-Context Services with apply_in_context — cross-context services via `[apply_in_context]` annotation (Tutorial 46).

Compiling and Running Programs at Runtime — compiling and running programs at runtime (Tutorial 44).

Contexts — language reference for context semantics.

Previous tutorial: *Compiling and Running Programs at Runtime*

Next tutorial: *Cross-Context Services with apply_in_context*

5.1.46 Cross-Context Services with apply_in_context

This tutorial covers `[apply_in_context]` — a macro annotation that rewrites functions so their bodies execute in a named debug agent context, while callers invoke them with normal function syntax.

Prerequisites: *Debug Agents* (Tutorial 45) for `fork_debug_agent_context` and `install_new_debug_agent`.

```
options gen2

require daslib/apply_in_context
require debugapi
```

Setting up a named context

First, create a debug agent context to host shared state. A plain `DapiDebugAgent` with no overrides is sufficient — the agent exists solely to own a named context:

```
var counter : int = 0

def install_service(ctx : Context) {
    install_new_debug_agent(new DapiDebugAgent(), "counter_service")
}

def init_counter_service() {
```

(continues on next page)

(continued from previous page)

```
    if (!has_debug_agent_context("counter_service")) {
        fork_debug_agent_context(@@install_service)
    }
}
```

The [apply_in_context] annotation

Functions annotated with `[apply_in_context(agent_name)]` have their body rewritten to execute in the named agent context. From the caller's perspective, these look like normal functions:

```
[apply_in_context(counter_service)]
def increment() : int {
    counter++
    return counter
}

[apply_in_context(counter_service)]
def get_counter() : int {
    return counter
}

[apply_in_context(counter_service)]
def add_to_counter(amount : int) {
    counter += amount
}

init_counter_service()
print("  increment() = {increment()}\n")
print("  increment() = {increment()}\n")
print("  get_counter() = {get_counter()}\n")
add_to_counter(10)
print("  after add = {get_counter()}\n")
print("  local counter = {counter}\n")
// output:
//   increment() = 1
//   increment() = 2
//   get_counter() = 2
//   after add = 12
//   local counter = 0
```

The agent context's `counter` is modified — the caller's local copy stays at zero.

Argument constraints

Arguments that cross context boundaries must use types that can be safely marshalled. Reference-type arguments must be marked `implicit`:

- `string implicit` — strings are reference types in daslang
- `var x :  int& implicit` — explicit reference parameters

Value types (`int`, `float`, `bool`) work without annotation:

```
[apply_in_context(counter_service)]
def set_counter_name(name : string implicit) {
    print("  counter named '{name}', value = {counter}\n")
}

[apply_in_context(counter_service)]
def read_counter(var result : int& implicit) {
    result = counter
}

set_counter_name("my_counter")
// output:
//   counter named 'my_counter', value = 12

var val = 0
read_counter(val)
print("  read_counter() -> val = {val}\n")
// output:
//   read_counter() -> val = 12
```

A cache service

A practical use case: a shared cache backed by a table that lives in the agent context. Any module or context can call put / get / has without worrying about which context they're in:

```
var cache : table<string; int>

def install_cache(ctx : Context) {
    install_new_debug_agent(new DapiDebugAgent(), "my_cache")
}

def init_cache_service() {
    if (!has_debug_agent_context("my_cache")) {
        fork_debug_agent_context(@@install_cache)
    }
}

[apply_in_context(my_cache)]
def cache_put(key : string implicit; value : int) {
    cache |> insert(key, value)
}

[apply_in_context(my_cache)]
```

(continues on next page)

(continued from previous page)

```
def cache_get(key : string implicit) : int {
    return cache?[key] ?? -1
}

[apply_in_context(my_cache)]
def cache_has(key : string implicit) : bool {
    return key_exists(cache, key)
}

[apply_in_context(my_cache)]
def cache_size() : int {
    return length(cache)
}

init_cache_service()
cache_put("width", 1920)
cache_put("height", 1080)
print("  cache size = {cache_size()}\n")
print("  width = {cache_get("width")}\n")
print("  local cache length = {length(cache)}\n")
// output:
//   cache size = 2
//   width = 1920
//   local cache length = 0
```

How it works under the hood

The `[apply_in_context]` macro rewrites each function into three parts:

1. **Caller function** (keeps original name) — verifies the agent exists, creates a result variable (if needed), then calls `invoke_in_context` to dispatch into the agent context.
2. **CONTEXT`func_name** (runs in the agent context) — verifies it's in the correct context, then calls the clone.
3. **CONTEXT_CLONE`func_name** (the original body) — contains the implementation code.

For a function with a return value, the expansion is similar to:

```
def get_counter() : int {
    verify(has_debug_agent_context("counter_service"))
    var __res__ : int
    unsafe {
        invoke_in_context(
            get_debug_agent_context("counter_service"),
            "counter_service`get_counter",
            addr(__res__)
        )
    }
    return __res__
}
```

The annotation adds `[pinvoke]` to the generated context function automatically, enabling the context mutex required for cross-context calls.

Quick reference

`[apply_in_context(name)]`	Rewrite function to execute in named agent context
`string implicit`	Required for string arguments crossing contexts
`var x : T& implicit`	Required for reference arguments crossing contexts
`require daslib/apply_in_context`	Import the annotation macro
`fork_debug_agent_context` + `install_new_...`	Set up the named context (see Tutorial 45)

See also:

Full source: `tutorials/language/46_apply_in_context.das`

Debug Agents — creating and installing debug agents (Tutorial 45).

Compiling and Running Programs at Runtime — `invoke_in_context` basics (Tutorial 44).

Contexts — language reference for context semantics.

Previous tutorial: *Debug Agents*

Next tutorial: *Data Walking with DapiDataWalker*

5.1.47 Data Walking with DapiDataWalker

This tutorial covers `DapiDataWalker` — a visitor-pattern class for inspecting and transforming daslang values at runtime. You subclass it, override callbacks for the types you care about, and walk any data through RTTI type information.

Prerequisites: basic daslang knowledge (structs, arrays, tables, enumerations, bitfields).

```
options gen2
options rtti

require rtti
require debugapi
require daslib/strings_boost
```

Minimal walker — scalar types

A `DapiDataWalker` subclass overrides only the callbacks you need. All 87 methods default to no-ops, so a minimal walker that prints integers, floats, strings, and booleans is very small:

```
class ScalarPrinter : DapiDataWalker {
    def override Int(var value : int&) : void {
        print("  int: {value}\n")
    }
    def override Float(var value : float&) : void {
        print("  float: {value}\n")
    }
    def override String(var value : string&) : void {
        print("  string: \"{value}\"\n")
    }
    def override Bool(var value : bool&) : void {
        print("  bool: {value}\n")
```

(continues on next page)

(continued from previous page)

```
    }
}
```

To walk a value, create the walker, wrap it with `make_data_walker`, then call `walk_data` with a pointer and `TypeInfo`:

```
var walker = new ScalarPrinter()
var inscope adapter <- make_data_walker(walker)

var x = 42
unsafe {
    adapter |> walk_data(addr(x), typeinfo rtti_typeinfo(x))
}
```

`typeinfo rtti_typeinfo(variable)` is a compile-time intrinsic that returns the RTTI `TypeInfo` for any variable or type expression.

Walking structures

The walker calls `beforeStructure/afterStructure` around the whole struct, and `beforeStructureField/afterStructureField` around each field. `StructInfo.name` gives the struct type name, `VarInfo.name` gives the field name:

```
struct Vec3 {
    x : float
    y : float
    z : float
}

struct Player {
    name : string
    health : int
    pos : Vec3
}

class StructPrinter : DapiDataWalker {
    indent : int = 0

    def pad() {
        for (_ in range(indent)) {
            print("  ")
        }
    }

    def override beforeStructure(ps : void?; si : StructInfo) : void {
        print("{si.name} \{\n")
        indent++
    }

    def override afterStructure(ps : void?; si : StructInfo) : void {
        indent--
        self->pad()
```

(continues on next page)

(continued from previous page)

```
        print("\}\n")
    }

    def override beforeStructureField(ps : void?; si : StructInfo;
            pv : void?; vi : VarInfo; last : bool) : void {
        self->pad()
        print("{vi.name} = ")
    }

    // ... plus Int, Float, String overrides
}
```

Walking a `Player` value produces indented output showing nested structures:

```
Player {
  name = "Alice"
  health = 100
  pos = Vec3 {
    x = 1
    y = 2.5
    z = -3
  }
}
```

Arrays and tables

Dynamic arrays trigger `beforeArrayData/afterArrayData` plus per-element callbacks. Tables trigger `beforeTable/afterTable` with key/value pairs:

```
class ContainerPrinter : DapiDataWalker {
    indent : int = 0

    def override beforeArrayData(ps : void?; stride : uint;
            count : uint; ti : TypeInfo) : void {
        self->pad()
        print("array[{int(count)}] = [\n")
        indent++
    }

    def override beforeArrayElement(ps : void?; ti : TypeInfo;
            pe : void?; index : uint; last : bool) : void {
        self->pad()
        print("[{int(index)}] = ")
    }

    def override beforeTable(pa : DapiTable; ti : TypeInfo) : void {
        self->pad()
        print("table[{int(pa.size)}] = \{\n")
        indent++
    }
```

(continues on next page)

(continued from previous page)

```
    // ... afterTable, beforeTableKey, afterTableKey, etc.
}
```

Note that `count`, `index`, and `pa.size` are `uint` values which print as hexadecimal by default — cast to `int` for decimal output.

Tuples and variants

Tuples walk each element by index. Variants walk only the active alternative. `beforeTupleEntry` receives the element index, `beforeVariant` receives the `TypeInfo` of the active case:

```
typedef Result = variant<ok : int; err : string>

class TupleVariantPrinter : DapiDataWalker {
    def override beforeTupleEntry(ps : void?; ti : TypeInfo;
            pv : void?; idx : int; last : bool) : void {
        self->pad()
        print("_{idx} = ")
    }

    def override beforeVariant(ps : void?; ti : TypeInfo) : void {
        self->pad()
        print("variant = ")
    }

    // ... plus scalar overrides for Int, String, etc.
}
```

Walking `Result(ok = 42)` prints `variant = 42`. Walking `Result(err = "not found")` prints `variant = "not found"`.

Enumerations and bitfields

Enumerations trigger `WalkEnumeration` with an `EnumInfo` containing value names. Bitfields trigger `Bitfield` with a `TypeInfo` containing field names:

```
class EnumBitfieldPrinter : DapiDataWalker {
    def override WalkEnumeration(var value : int&; ei : EnumInfo) : void {
        for (i in range(ei.count)) {
            unsafe {
                if (int(ei.fields[i].value) == value) {
                    print("  enum {ei.name}.{ei.fields[i].name} ({value})\n")
                    return
                }
            }
        }
        print("  enum {ei.name} = {value}\n")
    }

    def override Bitfield(var value : uint&; ti : TypeInfo) : void {
        print("  bitfield = {value} [")
```

(continues on next page)

(continued from previous page)

```
        var first = true
        for (i in range(ti.argCount)) {
            if ((value & (1u << uint(i))) != 0u) {
                if (!first) {
                    print(", ")
                }
                first = false
                unsafe {
                    print("{ti.argNames[i]}")
                }
            }
        }
        print("]\n")
    }
}
```

JSON serializer — putting it all together

A practical walker that serializes any daslang value to JSON. This combines structure, array, table, tuple, and scalar callbacks into one coherent class. The walker writes to a `StringBuilderWriter` for efficiency — no intermediate string concatenation:

```
class JsonWalker : DapiDataWalker {
    @do_not_delete writer : StringBuilderWriter?
    indent : int = 0
    needComma : array<bool>

    // --- structures ---
    def override beforeStructure(ps : void?; si : StructInfo) : void {
        *writer |> write("\{")
        indent++
        pushComma()
    }

    def override beforeStructureField(ps : void?; si : StructInfo;
            pv : void?; vi : VarInfo; last : bool) : void {
        comma()
        nl()
        *writer |> write("\"")
        *writer |> write(vi.name)
        *writer |> write("\": ")
    }

    // --- arrays ---
    def override beforeArrayData(ps : void?; stride : uint;
            count : uint; ti : TypeInfo) : void {
        *writer |> write("[")
        indent++
        pushComma()
    }
```

(continues on next page)

(continued from previous page)

```
    // ... tables, tuples, scalars
}
```

The `to_json` helper wraps the walk in `build_string`:

```
def to_json(var value; tinfo : TypeInfo) : string {
    var walker = new JsonWalker()
    var inscope adapter <- make_data_walker(walker)
    let res = build_string() $(var writer) {
        unsafe {
            walker.writer = addr(writer)
            adapter |> walk_data(addr(value), tinfo)
        }
    }
    unsafe { delete walker }
    return res
}
```

Walking an `Inventory` struct produces well-formatted JSON:

```
{
  "owner": "Alice",
  "gold": 250,
  "items": [
    {
      "name": "Sword",
      "weight": 3.5
    },
    {
      "name": "Shield",
      "weight": 5.2
    },
    {
      "name": "Potion",
      "weight": 0.5
    }
  ],
  "flags": {
    "is_merchant": true,
    "is_hostile": false
  }
}
```

Filtering with canVisit

All `canVisit*` methods return `true` by default. Override them to return `false` and the walker skips that subtree entirely:

```
class FilteringWalker : DapiDataWalker {
    skipStructName : string

    def override canVisitStructure(ps : void?; si : StructInfo) : bool {
        if (si.name == skipStructName) {
            print("<skipped {si.name}>\n")
            return false
        }
        return true
    }

    // ... beforeStructure, afterStructure, field and scalar overrides
}
```

Setting `skipStructName = "Secret"` skips the `Secret` struct entirely, including all its fields:

```
PublicRecord {
  title = "Performance Review"
  secret = <skipped Secret>
  score = 95
}
```

Mutating data in-place

Scalar callbacks receive `var value : T&` — a mutable reference. This means the walker can modify data during traversal:

```
class FloatClamper : DapiDataWalker {
    lo : float = 0.0
    hi : float = 1.0

    def override Float(var value : float&) : void {
        if (value < lo) {
            value = lo
        }
        if (value > hi) {
            value = hi
        }
    }
}
```

Walking a `Particle` with out-of-range floats clamps them to `[0..1]`:

```
Before clamping: x=-0.5 y=0.3 z=1.7 alpha=0.8
After clamping:  x=0 y=0.3 z=1 alpha=0.8
```

Quick reference

`DapiDataWalker`	Base class — subclass and override callbacks
`make_data_walker(walker)`	Wrap class instance into `smart_ptr<DataWalker>`
`walk_data(adapter, ptr, typeinfo)`	Walk data at `ptr` using RTTI `TypeInfo`
`typeinfo rtti_typeinfo(var)`	Get `TypeInfo` for any variable (compile-time)
`unsafe { addr(var) }`	Get `void?` pointer to a variable
`canVisit*(...)`	Return `false` to skip a subtree
`beforeX(...) / afterX(...)`	Container enter/exit callbacks
`Int, Float, String, Bool, ...`	Scalar value callbacks (mutable references)
`WalkEnumeration(value, EnumInfo)`	Enumeration callback with value metadata
`Bitfield(value, TypeInfo)`	Bitfield callback with field name metadata

See also:

Full source: `tutorials/language/47_data_walker.das`

stdlib_debugapi — `DapiDataWalker` class reference and `make_data_walker` / `walk_data` functions.

stdlib_rtti — RTTI types (`TypeInfo`, `StructInfo`, `VarInfo`, `EnumInfo`).

Previous tutorial: *Cross-Context Services with apply_in_context*

Next tutorial: *Compile-Time Field Iteration with apply*

5.1.48 Compile-Time Field Iteration with apply

This tutorial covers `daslib/apply` — a call macro that iterates struct, tuple, and variant fields at compile time. Unlike runtime RTTI walkers, `apply` generates specialized code per field with zero reflection overhead.

Prerequisites: basic daslang knowledge (structs, tuples, variants).

```
options gen2
options rtti

require rtti
require daslib/apply
require daslib/strings_boost
```

Basic struct iteration

`apply(value) $(name, field) { ... }` visits every field of a struct. `name` is a compile-time string constant with the field name, and `field` is the field value with its concrete type:

```
struct Hero {
    name   : string
    health : int
    speed  : float
}

let hero = Hero(name = "Archer", health = 100, speed = 3.5)

apply(hero) $(name, field) {
```

(continues on next page)

(continued from previous page)

```
    print("  {name} = {field}\n")
}
```

Output:

```
name = Archer
health = 100
speed = 3.5
```

Compile-time dispatch with static_if

Because `name` is known at compile time, `static_if` can branch on it. Only the matching branch compiles for each field — the others are discarded entirely:

```
struct Config {
    width      : int
    height     : int
    title      : string
    fullscreen : bool
}

let cfg = Config(width = 1920, height = 1080,
                 title = "My Game", fullscreen = true)

apply(cfg) $(name, field) {
    static_if (name == "title") {
        print("  Title (special handling): \"{field}\"\n")
    } else {
        print("  {name} = {field}\n")
    }
}
```

You can also dispatch on the **type** with `typeinfo stripped_typename(field)`.

Mutating fields

Pass a `var` (mutable) variable and `apply` gives mutable references to each field:

```
struct Stats {
    attack  : int
    defense : int
    magic   : int
}

var stats = Stats(attack = 10, defense = 5, magic = 8)

apply(stats) $(name, field) {
    field *= 2
}
// stats is now Stats(attack=20, defense=10, magic=16)
```

Tuples

apply works on tuples too. Unnamed tuple fields are named _0, _1, etc. Named tuples use their declared names:

```
let pair : tuple<int; string> = (42, "hello")
apply(pair) $(name, field) {
    print("  {name} = {field}\n")
}
// _0 = 42
// _1 = hello

let point : tuple<x : float; y : float> = (1.0, 2.0)
apply(point) $(name, field) {
    print("  {name} = {field}\n")
}
// x = 1
// y = 2
```

Variants

For variants, apply visits **only the currently active alternative**. The block fires for the one alternative that is set:

```
variant Shape {
    circle   : float
    rect     : float2
    triangle : float3
}

let s = Shape(circle = 5.0)
apply(s) $(name, field) {
    print("  Shape is {name}: {field}\n")
}
// Shape is circle: 5
```

Generic describe function

apply is ideal for building generic utilities that work on any struct without knowing its fields in advance:

```
def describe(value) {
    var first = true
    print("\{")
    apply(value) $(name, field) {
        if (!first) {
            print(", ")
        }
        first = false
        static_if (typeinfo stripped_typename(field) == "string") {
            print("{name}=\"{field}\"")
        } else {
            print("{name}={field}")
        }
```

(continues on next page)

(continued from previous page)

```
    }
    print("\}")
}
```

This prints any struct in `{field=value, ...}` format without writing type-specific code.

Field annotations (3-argument form)

Struct fields can carry metadata via `@` annotations:

```
struct DbRecord {
    @column="user_name"  name  : string
    @column="user_email" email : string
    @skip                id    : int
    @column="age"        age   : int
}
```

Annotation syntax:

- `@name` — boolean (defaults to `true`)
- `@name=value` — integer, float, or bare identifier (string)
- `@name="text"` — quoted string

The 3-argument form `apply(value) $(name, field, annotations)` receives annotations as `array<tuple<name:string; data:RttiValue>>` for each field. `RttiValue` is a variant with alternatives `tBool`, `tInt`, `tFloat`, `tString`, etc.

```
apply(record) $(name : string; field; annotations) {
    var column_name = name
    var skip = false
    for (ann in annotations) {
        if (ann.name == "skip") {
            skip = true
        } elif (ann.name == "column") {
            column_name = ann.data as tString
        }
    }
    if (!skip) {
        // use column_name and field ...
    }
}
```

This pattern powers `daslib/json_boost`'s `@rename`, `@optional`, `@enum_as_int`, `@unescape`, and `@embed` field annotations.

Full source

The complete tutorial source is in `tutorials/language/48_apply.das`.

Run it with:

```
daslang.exe tutorials/language/48_apply.das
```

See also:

Full source: `tutorials/language/48_apply.das`

stdlib_apply — `apply` call macro reference.

stdlib_rtti — `RttiValue` variant type used by the 3-argument annotation form.

Data Walking with DapiDataWalker — runtime data walking with `DapiDataWalker` (Tutorial 47).

Previous tutorial: *Data Walking with DapiDataWalker*

Next tutorial: *Async / Await*

5.1.49 Async / Await

This tutorial covers `daslib/async_boost` — an async/await framework built on top of daslang generators. Every `[async]` function is transformed at compile time into a state-machine generator. No threads, channels, or job queues are involved — everything is single-threaded cooperative multitasking.

Prerequisites: Tutorial 15 (Iterators and Generators), Tutorial 40 (Coroutines).

```
options gen2
options no_unused_function_arguments = false

require daslib/async_boost
require daslib/coroutines
```

Void async — the simplest form

An `[async]` function with `: void` return type becomes a `generator<bool>` state machine, just like a `[coroutine]`. Call `await_next_frame()` to suspend until the next step:

```
[async]
def greet(name : string) : void {
    print("  hello, ")
    await_next_frame()
    print("{name}!\n")
}

def demo_void_async() {
    print("=== void async ===\n")
    var it <- greet("world")
    var step = 1
    for (running in it) {
        print("  -- step {step} --\n")
        step ++
    }
```

(continues on next page)

(continued from previous page)

```
    print("  -- done --\n")
}
```

Each iteration of the `for` loop advances the generator by one step. The function body resumes after the last `await_next_frame()` and runs until the next one (or until it returns).

Typed async — yielding values

An `[async]` function with a non-void return type yields values wrapped in `variant<res:T; wait:bool>`. Each `await_next_frame()` yields `variant(wait=true)`; each `yield value` yields `variant(res=value)`:

```
[async]
def compute(x : int) : int {
    await_next_frame()  // simulate one frame of work
    yield x * 2
}

def demo_typed_async() {
    for (v in compute(21)) {
        if (v is wait) {
            print("  (waiting...)\n")
        } elif (v is res) {
            print("  result = {v as res}\n")  // 42
        }
    }
}
```

The consumer checks `v is wait` vs `v is res` to distinguish suspension frames from actual results.

Await — waiting for an async result

Inside an `[async]` function you can `await` another async call. `await` suspends the parent until the child completes and extracts the result:

```
[async]
def add_one(x : int) : int {
    await_next_frame()
    yield x + 1
}

[async]
def chained_math() : void {
    var a = 0
    a = await <| add_one(0)     // copy-assign:  a = 1
    a <- await <| add_one(a)    // move-assign:  a = 2
    let b <- await <| add_one(a) // let-bind:    b = 3
    print("  a={a}, b={b}\n")
}
```

Three forms of `await`:

- `a = await <| fn(args)` — copy-assign the result

- `a <- await <| fn(args)` — move-assign the result
- `let b <- await <| fn(args)` — bind to a new variable

Struct return with move semantics

Async functions can yield structs. Use `yield <-` to move the result out (useful for non-copyable data):

```
struct Measurement {
    sensor_id : int
    value : float
    tag : string
}

[async]
def read_sensor(id : int) : Measurement {
    await_next_frame()
    var m : Measurement
    m.sensor_id = id
    m.value = 3.14
    m.tag = "temperature"
    yield <- m
}

[async]
def process_sensors() : void {
    let m <- await <| read_sensor(1)
    print("  sensor {m.sensor_id}: {m.value} ({m.tag})\n")
}
```

Iterating async generators

A typed async function that yields multiple values acts as an asynchronous generator. Consumers iterate and check `v is res` to extract each value:

```
[async]
def fibonacci_async(n : int) : int {
    var a = 0
    var b = 1
    for (i in range(n)) {
        yield a
        let next_val = a + b
        a = b
        b = next_val
        await_next_frame()
    }
}

def demo_async_iteration() {
    print("  fibonacci: ")
    for (v in fibonacci_async(8)) {
        if (v is res) {
```

(continues on next page)

(continued from previous page)

```
            print("{v as res} ")
        }
    }
    print("\n")
}
```

Output: `0 1 1 2 3 5 8 13`

Running tasks

`async_boost` provides four task runners:

- `async_run(it)` — drive a single task to completion
- `async_run_all(tasks)` — drive all tasks cooperatively, round-robin one step per task per frame
- `async_timeout(it, max_frames)` — drive a task for at most *max_frames* steps; returns `true` if it completed in time
- `async_race(a, b)` — drive two tasks cooperatively; returns `0` if *a* finishes first, `1` if *b* finishes first

```
[async]
def worker(name : string; frames : int) : void {
    for (i in range(frames)) {
        print("  {name}: frame {i + 1}/{frames}\n")
        await_next_frame()
    }
}
```

`async_run` steps through a single task:

```
var single <- worker("solo", 3)
async_run(single)
```

`async_run_all` interleaves multiple tasks:

```
var tasks : array<iterator<bool>>
tasks |> emplace <| worker("alpha", 2)
tasks |> emplace <| worker("beta", 3)
async_run_all(tasks)
```

`async_timeout` enforces a deadline:

```
var fast <- worker("fast", 2)
let completed = async_timeout(fast, 10)   // true
var slow <- worker("slow", 100)
let timed_out = async_timeout(slow, 3)    // false (timeout)
```

`async_race` runs two tasks and returns which finishes first:

```
var racer_a <- worker("A", 2)
var racer_b <- worker("B", 5)
let winner = async_race(racer_a, racer_b)  // 0 (A wins)
```

Mixing async with coroutines

An [async] function can await a [coroutine]. This lets you compose low-level coroutine logic with high-level async orchestration:

```
[coroutine]
def tick_counter(n : int) {
    for (i in range(n)) {
        print("  tick {i + 1}\n")
        co_continue()
    }
}

[async]
def orchestrator() : void {
    print("  orchestrator: start\n")
    await_next_frame()
    print("  orchestrator: awaiting coroutine\n")
    await <| tick_counter(3)
    print("  orchestrator: coroutine done\n")
    await_next_frame()
    print("  orchestrator: finish\n")
}
```

Full source

The complete tutorial source is in tutorials/language/49_async.das.

Run it with:

```
daslang.exe tutorials/language/49_async.das
```

See also:

Full source: tutorials/language/49_async.das

stdlib_async_boost — async_boost module reference.

stdlib_coroutines — coroutines module reference (underlying generator framework).

Coroutines — Tutorial 40: Coroutines (prerequisite).

Previous tutorial: *Compile-Time Field Iteration with apply*

Next tutorial: *Structure-of-Arrays (SOA)*

5.1.50 Structure-of-Arrays (SOA)

This tutorial covers daslib/soa — a compile-time macro that transforms regular structs into a Structure-of-Arrays layout. The [soa] annotation generates parallel arrays for every field, plus all the container operations you need (push, erase, pop, clear, resize, reserve, swap, from_array, to_array).

Prerequisites: familiarity with structs and arrays.

```
options gen2
options no_unused_function_arguments = false

require daslib/soa
```

What is SOA?

Normally a struct is stored as Array-of-Structures (AOS):

```
[ {x,y,z,w}, {x,y,z,w}, {x,y,z,w}, ... ]
```

Structure-of-Arrays (SOA) rearranges this into parallel arrays:

```
xs: [x, x, x, ...]
ys: [y, y, y, ...]
...
```

This is friendlier to CPU caches when iterating over a single field (e.g. updating positions), because the data is contiguous.

The `[soa]` annotation automates this transformation. For a struct `Particle` with fields `pos`, `vel`, `life`, `color`, it generates `Particle`SOA` where each field is an `array<FieldType>`.

Basic SOA

Annotate a struct with `[soa]` and the macro generates the SOA layout plus all access functions:

```
[soa]
struct Particle {
    pos   : float3
    vel   : float3
    life  : float
    color : float4
}
```

Declare and use the SOA container:

```
def demo_basic_soa() {
    print("=== basic SOA ===\n")
    var particles : Particle`SOA

    particles |> push <| Particle(
        pos = float3(0.0), vel = float3(1.0, 0.0, 0.0),
        life = 3.0, color = float4(1.0, 0.0, 0.0, 1.0))
    particles |> push <| Particle(
        pos = float3(5.0), vel = float3(0.0, 1.0, 0.0),
        life = 5.0, color = float4(0.0, 1.0, 0.0, 1.0))
    particles |> push <| Particle(
        pos = float3(10.0), vel = float3(0.0, 0.0, 1.0),
        life = 2.0, color = float4(0.0, 0.0, 1.0, 1.0))
    print("  count: {length(particles)}\n")
```

(continues on next page)

(continued from previous page)

```
    // Indexed access - macro rewrites soa[i].field to soa.field[i]
    print("  particles[0].pos  = {particles[0].pos}\n")
    print("  particles[1].life = {particles[1].life}\n")
    print("  particles[2].vel  = {particles[2].vel}\n")
}
```

Iteration

The `SoaForLoop` macro transforms `for (it in soa)` into a multi-source loop over the individual column arrays. Only the fields you actually access are iterated:

```
def demo_iteration() {
    print("\n=== iteration ===\n")
    var particles : Particle`SOA
    for (i in range(4)) {
        particles |> push <| Particle(
            pos = float3(float(i)), life = float(4 - i))
    }

    for (it in particles) {
        print("    pos={it.pos} life={it.life}\n")
    }

    // Mixed iteration with an index counter
    for (idx, it in count(), particles) {
        print("    [{idx}] pos={it.pos}\n")
    }
}
```

Container operations

`push`, `push_clone`, `emplace`, `erase`, `pop`, and `clear` all work the same as on regular arrays:

```
def demo_container_ops() {
    print("\n=== container operations ===\n")
    var soa : Particle`SOA

    // push - move semantics
    soa |> push <| Particle(pos = float3(1.0), life = 10.0)
    soa |> push <| Particle(pos = float3(2.0), life = 20.0)
    soa |> push <| Particle(pos = float3(3.0), life = 30.0)

    // push_clone - copy from a const value
    let p = Particle(pos = float3(4.0), life = 40.0)
    soa |> push_clone(p)

    // erase - remove by index
    soa |> erase(1)

    // pop - remove last element
```

(continues on next page)

(continued from previous page)

```
    soa |> pop()

    // clear - remove all elements
    soa |> clear()
}
```

Sizing — resize, reserve, capacity

Pre-allocate memory with `reserve`, query with `capacity`, and change the element count with `resize`:

```
def demo_sizing() {
    print("\n=== sizing ===\n")
    var soa : Particle`SOA

    soa |> reserve(100)
    print("  capacity={capacity(soa)}\n")

    for (i in range(10)) {
        soa |> push <| Particle(
            pos = float3(float(i)), life = float(i))
    }

    // resize - truncate or extend with defaults
    soa |> resize(5)
    soa |> resize(8)
}
```

Swap and sorting

`swap` exchanges all fields of two elements at once. Combine it with any sorting algorithm:

```
[soa]
struct SortItem {
    key : int
    tag : string
}

def demo_swap_and_sort() {
    var soa : SortItem`SOA
    soa |> push <| SortItem(key = 30, tag = "gamma")
    soa |> push <| SortItem(key = 10, tag = "alpha")
    soa |> push <| SortItem(key = 20, tag = "beta")

    soa |> swap(0, 2)

    // Bubble sort
    let n = length(soa)
    for (pass_idx in range(n)) {
        for (k in range(n - 1)) {
            if (soa[k].key > soa[k + 1].key) {
```

(continues on next page)

(continued from previous page)

```
                soa |> swap(k, k + 1)
            }
        }
    }
}
```

Bulk conversion — from_array, to_array

Convert between AOS (array<T>) and SOA (T`SOA) layouts:

```
[soa]
struct Vec2 {
    x : float
    y : float
}

def demo_conversion() {
    var arr : array<Vec2>
    arr |> push <| Vec2(x = 1.0, y = 2.0)
    arr |> push <| Vec2(x = 3.0, y = 4.0)
    arr |> push <| Vec2(x = 5.0, y = 6.0)

    // AOS → SOA
    var soa : Vec2`SOA
    soa |> from_array(arr)

    // SOA → AOS
    var result <- to_array(soa)
}
```

Particle simulation

A complete example: spawn particles, simulate physics, remove dead ones:

```
def demo_particle_sim() {
    var particles : Particle`SOA
    particles |> reserve(100)

    for (i in range(5)) {
        let fi = float(i)
        particles |> push <| Particle(
            pos   = float3(fi * 2.0, 0.0, 0.0),
            vel   = float3(0.0, 1.0 + fi * 0.5, 0.0),
            life  = 3.0 + fi,
            color = float4(fi / 4.0, 1.0 - fi / 4.0, 0.5, 1.0)
        )
    }

    let dt = 1.0
    for (step in range(3)) {
```

(continues on next page)

(continued from previous page)

```
        for (it in particles) {
            it.pos += it.vel * dt
            it.life -= dt
        }
        // Remove dead - iterate backwards
        var i = length(particles) - 1
        while (i >= 0) {
            if (particles[i].life <= 0.0) {
                particles |> erase(i)
            }
            i --
        }
    }
}
```

Game entity table

SOA works well for game entity tables with mixed field types:

```
[soa]
struct GameEntity {
    id     : int
    name   : string
    health : float
    alive  : bool
}

def demo_entity_table() {
    var entities : GameEntity`SOA
    entities |> push <| GameEntity(
        id = 1, name = "warrior", health = 100.0, alive = true)
    entities |> push <| GameEntity(
        id = 2, name = "mage",    health = 60.0,  alive = true)
    entities |> push <| GameEntity(
        id = 3, name = "archer",  health = 80.0,  alive = true)

    // Apply damage
    for (it in entities) {
        it.health -= 55.0
        if (it.health <= 0.0) {
            it.alive = false
        }
    }

    // Convert to AOS for serialization
    var snapshot <- to_array(entities)
}
```

Full source

The complete tutorial source is in `tutorials/language/50_soa.das`.

Run it with:

```
daslang.exe tutorials/language/50_soa.das
```

See also:

Full source: `tutorials/language/50_soa.das`

stdlib_soa — SOA module reference.

Previous tutorial: *Async / Await*

5.2 C Integration Tutorials

These tutorials show how to embed daslang in a C application using the `daScriptC.h` API. Each tutorial comes with a `.c` source file and a companion `.das` script in `tutorials/integration/c/`.

5.2.1 C Integration: Hello World

This tutorial shows how to embed daslang in a C application using the `daScriptC.h` pure-C API. By the end you will have a standalone C program that compiles a `.das` script, finds a function, calls it, and prints `Hello from daslang!`.

Note: The C API uses opaque handle types (pointers you never dereference) and `vec4f` for passing arguments. A higher-level C++ API is also available in `daScript.h`; this tutorial series focuses on the C API only.

Prerequisites

- daslang built from source (`cmake --build build --config Release`). The build produces `libDaScript` which the C tutorial links against.
- The `daScriptC.h` header — located in `include/daScript/daScriptC.h`.

The daslang file

Create a minimal script with a single exported function:

```
options gen2

[export]
def test() {
    print("Hello from daslang!\n")
}
```

The `[export]` annotation makes the function visible to the host so that `das_context_find_function` can locate it by name.

The C program

The program follows a strict lifecycle. Each step maps to one or two API calls.

Step 1 — Initialize

```
#include "daScript/daScriptC.h"

int main(int argc, char ** argv) {
    das_initialize();
```

`das_initialize` must be called **once** before any other API call. It registers all built-in modules (math, strings, etc.).

Step 2 — Create supporting objects

```
das_text_writer  * tout         = das_text_make_printer();
das_module_group * module_group = das_modulegroup_make();
das_file_access  * file_access  = das_fileaccess_make_default();
```

Object	Purpose
`das_text_writer`	Receives compiler/runtime messages (prints to stdout)
`das_module_group`	Holds modules available during compilation
`das_file_access`	Provides file I/O to the compiler (default = disk)

Step 3 — Compile the script

```
das_program * program = das_program_compile(
    script_path, file_access, tout, module_group);

if (das_program_err_count(program)) {
    // handle errors ...
}
```

`das_program_compile` reads the `.das` file, parses, type-checks, and produces a program object. Errors (if any) are written to `tout` and can also be iterated with `das_program_get_error`.

Step 4 — Create a context and simulate

```
das_context * ctx = das_context_make(
    das_program_context_stack_size(program));

if (!das_program_simulate(program, ctx, tout)) {
    // handle errors ...
}
```

A **context** is the runtime environment — it owns the execution stack, global variables, and heap. **Simulation** resolves function pointers, initializes globals, and prepares everything for execution.

Step 5 — Find the function

```
das_function * fn_test = das_context_find_function(ctx, "test");
if (!fn_test) {
    printf("function 'test' not found\n");
}
```

Any function marked `[export]` in the script can be found by name.

Step 6 — Call the function

```
das_context_eval_with_catch(ctx, fn_test, NULL);

char * exception = das_context_get_exception(ctx);
if (exception) {
    printf("exception: %s\n", exception);
}
```

`das_context_eval_with_catch` runs the function inside a try/catch so that daslang exceptions do not crash the host application. The third argument is an array of `vec4f` arguments — `NULL` here because `test` takes no parameters.

Step 7 — Clean up and shut down

```
    das_context_release(ctx);
    das_program_release(program);
    das_fileaccess_release(file_access);
    das_modulegroup_release(module_group);
    das_text_release(tout);

    das_shutdown();
    return 0;
}
```

Release objects in reverse order of creation. `das_shutdown` frees all global state — no API calls are allowed after it.

Key concepts

Opaque handles
: Every daslang object (program, context, function, etc.) is an **opaque handle** — a pointer whose internal layout is hidden from C. You create them with `das_*_make` / `das_*_create` and free them with `das_*_release`. Never cast or dereference them.

Lifecycle ownership
: The C host owns all handles it creates. `das_program_release` frees the program; `das_context_release` frees the context. Forgetting to release a handle leaks memory.

Error checking
: Always check `das_program_err_count` after compilation **and** after simulation. Always check `das_context_get_exception` after every `das_context_eval_*` call.

Building and running

The tutorial is built automatically by CMake as part of the daslang project:

```
cmake --build build --config Release --target integration_c_01
```

Run from the project root so that the script path resolves correctly:

```
bin\Release\integration_c_01.exe
```

Expected output:

```
Hello from daslang!
```

See also:

Full source: `01_hello_world.c`, `01_hello_world.das`

Next tutorial: tutorial_integration_c_calling_functions

type_mangling — how types are encoded as strings in the C API

daScriptC.h API header: `include/daScript/daScriptC.h`

5.2.2 C Integration: Calling daslang Functions

This tutorial shows how to pass arguments to daslang functions from C and retrieve their return values. It covers all the common scalar types (int, float, string, bool) and introduces the **complex result** (cmres) calling convention for functions that return structures.

Arguments and return values

Every value crossing the C ↔ daslang boundary is carried in a `vec4f` — a 128-bit SSE register. The API provides symmetric helper pairs:

Packing (C → daslang)	Unpacking (daslang → C)
`das_result_int(value)`	`das_argument_int(vec4f)`
`das_result_float(value)`	`das_argument_float(vec4f)`
`das_result_string(value)`	`das_argument_string(vec4f)`
`das_result_bool(value)`	`das_argument_bool(vec4f)`

Despite their names, the `das_result_*` helpers are also used to **pack arguments** into a `vec4f` array before calling a daslang function.

Calling a function with arguments

```
// Find the function (must be [export] in the script).
das_function * fn = das_context_find_function(ctx, "add");

// Pack two int arguments into a vec4f array.
vec4f args[2];
args[0] = das_result_int(17);
```

(continues on next page)

(continued from previous page)

```
args[1] = das_result_int(25);

// Call and unpack the int return value.
vec4f ret = das_context_eval_with_catch(ctx, fn, args);
int result = das_argument_int(ret);    // 42
```

The corresponding daslang function:

```
[export]
def add(a : int; b : int) : int {
    return a + b
}
```

String arguments and return values

Strings are passed as plain `char *` pointers. When daslang **returns** a string, it lives on the context's heap and remains valid until the context is released:

```
vec4f args[1];
args[0] = das_result_string("World");

vec4f ret = das_context_eval_with_catch(ctx, fn_greet, args);
char * greeting = das_argument_string(ret);
printf("%s\n", greeting);   // "Hello, World!"
```

Returning structures (complex results)

When a daslang function returns a struct, use `das_context_eval_with_catch_cmres`. You allocate the struct on the C side and pass a pointer:

```
typedef struct { float x; float y; } Vec2;

vec4f args[2];
args[0] = das_result_float(1.5f);
args[1] = das_result_float(2.5f);

Vec2 v;
das_context_eval_with_catch_cmres(ctx, fn_make_vec2, args, &v);
printf("x=%.1f y=%.1f\n", v.x, v.y);   // x=1.5 y=2.5
```

The C struct layout **must** match the daslang struct exactly (same field order, same types, same padding).

Building and running

```
cmake --build build --config Release --target integration_c_02
bin\Release\integration_c_02.exe
```

Expected output:

```
add(17, 25) = 42
square(3.5) = 12.25
greet("World") = "Hello, World!"
int=42 float=3.14 string=test bool=true
make_vec2(1.5, 2.5) = { x=1.5, y=2.5 }
```

See also:

Full source: `02_calling_functions.c`, `02_calling_functions.das`

Previous tutorial: tutorial_integration_c_hello_world

Next tutorial: tutorial_integration_c_binding_types

type_mangling — complete type mangling reference

5.2.3 C Integration: Binding C Types

This tutorial shows how to create a **custom module** in C and expose C types (enumerations, structures, aliases) to daslang so that scripts can use them as native types.

Creating a module

A module is a named container for types and functions. The script refers to it with `require`:

```
das_module * mod = das_module_create("tutorial_c_03");
```

The module must be registered **before** compiling any script that `require`s it. After `das_initialize()`, call your registration function before `das_program_compile`.

Binding an enumeration

```
// C enum
typedef enum { Color_red = 0, Color_green = 1, Color_blue = 2 } Color;

// Bind to daslang
das_enumeration * en = das_enumeration_make("Color", "Color", 1);
das_enumeration_add_value(en, "red",   "Color_red",   Color_red);
das_enumeration_add_value(en, "green", "Color_green", Color_green);
das_enumeration_add_value(en, "blue",  "Color_blue",  Color_blue);
das_module_bind_enumeration(mod, en);
```

The third argument to `das_enumeration_make` selects the underlying integer type: `0` = int8, `1` = int32, `2` = int16.

In daslang:

```
let color = Color.green
print("color = {color}\n")
```

Binding a structure

C structures are exposed as **handled types**. You specify size, alignment, and each field's offset and mangled type:

```
typedef struct { float x; float y; } Point2D;

das_structure * st = das_structure_make(lib, "Point2D", "Point2D",
                                        sizeof(Point2D), _Alignof(Point2D));
das_structure_add_field(st, mod, lib, "x", "x", offsetof(Point2D, x), "f");
das_structure_add_field(st, mod, lib, "y", "y", offsetof(Point2D, y), "f");
das_module_bind_structure(mod, st);
```

In daslang:

```
var p : Point2D
p.x = 3.0
p.y = 4.0
```

Type mangling reference

Every type in the C API is described by a compact string:

Mangled string	Type
`i`	`int`
`u`	`uint`
`f`	`float`
`d`	`double`
`b`	`bool`
`s`	`string`
`v`	`void`
`1<i>A`	`array<int>`
`1<f>?`	`float?` (pointer)
`H<Name>`	handled struct `Name`

See type_mangling for the complete specification.

Binding a type alias

```
das_module_bind_alias(mod, lib, "IntArray", "1<i>A");
```

In daslang, `IntArray` is now a synonym for `array<int>`.

Binding interop functions

Functions that operate on custom types use the standard interop pattern, but their mangled signatures reference the handled type by name:

```
// def point_distance(p : Point2D) : float
// Mangled: "f H<Point2D>"
vec4f c_point_distance(das_context * ctx, das_node * node, vec4f * args) {
    Point2D * p = (Point2D *)das_argument_ptr(args[0]);
    float dist = sqrtf(p->x * p->x + p->y * p->y);
    return das_result_float(dist);
}

das_module_bind_interop_function(mod, lib, &c_point_distance,
    "point_distance", "c_point_distance",
    SIDEEFFECTS_none, "f H<Point2D>");
```

Note the use of `das_argument_ptr` — structures are passed as pointers.

Building and running

```
cmake --build build --config Release --target integration_c_03
bin\Release\integration_c_03.exe
```

Expected output:

```
color = green
numbers = [10, 20, 30]
point = (3, 4)
distance from origin = 5
as string: (3.00, 4.00)
```

See also:

Full source: `03_binding_types.c`, `03_binding_types.das`

Previous tutorial: tutorial_integration_c_calling_functions

Next tutorial: tutorial_integration_c_callbacks

type_mangling — complete type mangling reference

5.2.4 C Integration: Callbacks and Closures

This tutorial shows how C code can **receive and invoke** daslang callable types: function pointers, lambdas, and blocks.

Three callable types

daslang has three callable types, each with different semantics:

Type	Mangled	Characteristics
function pointer	`@@`	No capture, cheapest, like a C function pointer
lambda	`@`	Heap-allocated, captures variables by reference
block	`$`	Stack-allocated, captures, cannot outlive scope

The mangled signature encodes the callable type after the argument list:

- `0<i;f>@@` — function pointer taking `(int, float)`
- `0<i;f>@` — lambda taking `(int, float)`
- `0<i;f>$` — block taking `(int, float)`

A leading return type is prepended: `"s 0<i;f>@@ i f"` means *returns string, takes function<(int,float):string>, int, float*.

Calling a function pointer

```
vec4f c_call_function(das_context * ctx, das_node * node, vec4f * args) {
    das_function * callback = das_argument_function(args[0]);
    int a    = das_argument_int(args[1]);
    float b  = das_argument_float(args[2]);

    vec4f cb_args[2];
    cb_args[0] = das_result_int(a);
    cb_args[1] = das_result_float(b);

    vec4f ret = das_context_eval_with_catch(ctx, callback, cb_args);
    return ret;
}
```

From daslang, pass a function pointer with `@@`:

```
def format_result(a : int; b : float) : string {
    return "formatted: {a} + {b}"
}

let r = c_call_function(@@format_result, 10, 2.5)
```

Calling a lambda

Lambdas carry captured state. When calling from C, **the first argument must be the lambda itself** (its capture block):

```
vec4f c_call_lambda(das_context * ctx, das_node * node, vec4f * args) {
    das_lambda * lambda = das_argument_lambda(args[0]);
    int a    = das_argument_int(args[1]);
    float b  = das_argument_float(args[2]);

    // First arg = lambda capture, then actual arguments.
```

(continues on next page)

(continued from previous page)

```
    vec4f lmb_args[3];
    lmb_args[0] = das_result_lambda(lambda);
    lmb_args[1] = das_result_int(a);
    lmb_args[2] = das_result_float(b);

    vec4f ret = das_context_eval_lambda(ctx, lambda, lmb_args);
    return ret;
}
```

Important: `das_context_eval_lambda` requires the lambda as `lmb_args[0]`. Omitting it will crash or produce garbage results.

From daslang:

```
var counter = 0
var lmb <- @(a : int; b : float) : string {
    counter += a
    return "lambda: a={a} b={b} counter={counter}"
}
let r = c_call_lambda(lmb, 5, 1.5)
```

Calling a block

Blocks are lighter than lambdas — they live on the stack and cannot outlive the scope that created them. Unlike lambdas, **block arguments do not include the block itself**:

```
vec4f c_call_block(das_context * ctx, das_node * node, vec4f * args) {
    das_block * block = das_argument_block(args[0]);
    int    a = das_argument_int(args[1]);
    float  b = das_argument_float(args[2]);

    vec4f blk_args[2];
    blk_args[0] = das_result_int(a);
    blk_args[1] = das_result_float(b);

    vec4f ret = das_context_eval_block(ctx, block, blk_args);
    return ret;
}
```

From daslang:

```
var blk <- $(a : int; b : float) : string {
    counter += a
    return "block: a={a} b={b} counter={counter}"
}
let r = c_call_block(blk, 7, 0.5)
```

Mangled signature cheat sheet

```
"s 0<i;f>@@ i f"   →  function pointer callback
"s 0<i;f>@  i f"   →  lambda callback
"s 0<i;f>$  i f"   →  block callback
```

Building and running

```
cmake --build build --config Release --target integration_c_04
bin\Release\integration_c_04.exe
```

Expected output:

```
--- function pointer callback ---
[C] calling function pointer with (10, 2.50)
[C] callback returned: "formatted: 10 + 2.5"
C returned: formatted: 10 + 2.5

--- lambda callback ---
[C] calling lambda with (5, 1.50)
[C] lambda returned: "lambda: a=5 b=1.5 counter=5"
C returned: lambda: a=5 b=1.5 counter=5
[C] calling lambda with (3, 0.50)
[C] lambda returned: "lambda: a=3 b=0.5 counter=8"
C returned: lambda: a=3 b=0.5 counter=8

--- block callback ---
[C] calling block with (7, 0.50)
[C] block returned: "block: a=7 b=0.5 counter=7"
C returned: block: a=7 b=0.5 counter=7
[C] calling block with (3, 1.50)
[C] block returned: "block: a=3 b=1.5 counter=10"
C returned: block: a=3 b=1.5 counter=10
```

See also:

Full source: `04_callbacks.c`, `04_callbacks.das`

Previous tutorial: tutorial_integration_c_binding_types

Next tutorial: tutorial_integration_c_unaligned_advanced

type_mangling — complete type mangling reference

5.2.5 C Integration: Unaligned ABI & Advanced

This tutorial covers three advanced topics: the **unaligned calling convention**, **inline script source**, and **detailed error reporting**.

The unaligned ABI

The default (aligned) ABI uses `vec4f` — a 128-bit SSE type that **must** be 16-byte aligned. This is fast but requires SSE support and aligned memory.

The **unaligned** ABI uses `vec4f_unaligned` — a plain struct of four floats with no alignment requirement:

```
typedef struct { float x, y, z, w; } vec4f_unaligned;
```

This makes it portable to platforms without SSE (ARM, WASM, etc.).

Aligned	Unaligned
`das_interop_function`	`das_interop_function_unaligned`
`das_context_eval_with_catch`	`das_context_eval_with_catch_unaligned`
`das_argument_int(args[i])`	`das_argument_int_unaligned(args + i)`
`das_result_int(value)`	`das_result_int_unaligned(result, value)`
`das_module_bind_interop_function`	`das_module_bind_interop_function_unaligned`

Unaligned interop function

```
// Signature differs: returns void, result is an out-parameter.
void c_greet(das_context * ctx, das_node * node,
             vec4f_unaligned * args, vec4f_unaligned * result) {
    int n = das_argument_int_unaligned(args + 0);
    printf("greet(%d)\n", n);
    das_result_void_unaligned(result);
}

// Bind with the _unaligned variant.
das_module_bind_interop_function_unaligned(mod, lib, &c_greet,
    "c_greet", "c_greet", SIDEEFFECTS_modifyExternal, "v i");
```

Calling with unaligned eval

```
vec4f_unaligned args[2];
das_result_int_unaligned(args + 0, 17);
das_result_int_unaligned(args + 1, 25);

vec4f_unaligned result;
das_context_eval_with_catch_unaligned(ctx, fn_add, args, 2, &result);
int sum = das_argument_int_unaligned(&result);
```

Note the extra `narguments` parameter (`2`) — the unaligned API needs the argument count explicitly.

Inline script source

Instead of loading a `.das` file from disk, you can embed the script as a C string and register it as a virtual file:

```
static const char * SCRIPT =
    "options gen2\n"
    "[export]\n"
    "def test() {\n"
    "    print(\"Hello from inline script!\\n\")\n"
    "}\n";

das_file_access * fa = das_fileaccess_make_default();
das_fileaccess_introduce_file(fa, "inline.das", SCRIPT, 0);

das_program * prog = das_program_compile("inline.das", fa, tout, lib);
```

The virtual file name (`"inline.das"`) can be anything — it never touches the file system. This is useful for:

- Embedded applications with no file system
- Testing and prototyping
- Configuration scripts shipped as resource data

Error reporting

When compilation fails, iterate the errors with `das_program_get_error` and extract human-readable messages:

```
int err_count = das_program_err_count(program);
for (int i = 0; i < err_count; i++) {
    das_error * error = das_program_get_error(program, i);

    // Option A: print to a text writer (stdout or string)
    das_error_output(error, tout);

    // Option B: extract into a C buffer
    char buf[1024];
    das_error_report(error, buf, sizeof(buf));
    printf("error: %s\n", buf);
}
```

Building and running

```
cmake --build build --config Release --target integration_c_05
bin\Release\integration_c_05.exe
```

Expected output (Part 1 — inline script + unaligned ABI):

```
=== Part 1: Inline script with unaligned ABI ===

Hello from inline script!
[C unaligned] greet called with n=42
add(17, 25) = 42
```

Expected output (Part 2 — error reporting):

```
=== Part 2: Error reporting ===

Compilation produced 1 error(s):
  error 0: bad_script.das:4:19: ...
```

See also:

Full source: `05_unaligned_advanced.c`, `05_unaligned_advanced.das`

Previous tutorial: tutorial_integration_c_callbacks

Next tutorial: tutorial_integration_c_sandbox

type_mangling — complete type mangling reference

daScriptC.h API header: `include/daScript/daScriptC.h`

5.2.6 C Integration: Sandbox

This tutorial demonstrates how to run daslang code in a **sandboxed environment** from a C host. The sandbox has two independent layers:

Layer	Controls	API
Filesystem lock	Which files exist (physical)	`das_fileaccess_introduce_*`, `das_fileaccess_lock`
Policy (.das_project)	What is allowed among existing files	`das_fileaccess_make_project`, callback functions

Filesystem locking

The filesystem lock prevents the compiler from accessing any file that was not **pre-introduced** into the file access cache:

```
das_file_access * fa = das_fileaccess_make_project(project_path);

// Pre-cache all standard library files
das_fileaccess_introduce_daslib(fa);

// Introduce user scripts as virtual files (inline C strings)
das_fileaccess_introduce_file(fa, "helpers.das", HELPER_SOURCE, 0);
das_fileaccess_introduce_file(fa, "script.das", SCRIPT_SOURCE, 0);

// Lock - getNewFileInfo() now returns null for uncached files
das_fileaccess_lock(fa);
```

The `0` in `das_fileaccess_introduce_file` means the file access **borrows** the string pointer (no copy) — suitable for static C string constants. Pass 1 for dynamically allocated strings that should be owned and freed by the file access.

Available `introduce` functions:

Function	Purpose
`das_fileaccess_introduce_file`	Register a virtual file from a string
`das_fileaccess_introduce_file_from_disk`	Read a disk file into cache
`das_fileaccess_introduce_daslib`	Cache all `daslib/*.das` files
`das_fileaccess_introduce_native_modules`	Cache all native plugin modules
`das_fileaccess_introduce_native_module`	Cache a single native module

The .das_project file

A `.das_project` is a regular daslang file that the compiler loads **before** compiling user scripts. It exports `[export]` callback functions that the compiler invokes during compilation.

The project file is loaded via `das_fileaccess_make_project`:

```
das_file_access * fa = das_fileaccess_make_project(
    "path/to/project.das_project");
```

Internally, the constructor:

1. Compiles the `.das_project` file as a normal daslang program
2. Simulates it in a fresh context
3. Looks up `[export]` functions by name (`module_get`, `module_allowed`, etc.)
4. Runs `[init]` functions
5. Sets the `DAS_PAK_ROOT` variable to the project file's directory

The project file itself is **not** subject to sandbox restrictions — it compiles with full filesystem access. Restrictions only apply to **subsequent** user script compilations.

Callback reference

The compiler recognizes these exported callback function names:

Function	Required	Purpose
`module_get(req, from : string) : module_info`	**Yes**	Resolve `require` paths to `(moduleName, fileName, importName)`
`module_allowed(mod, filename : string) : bool`	No	Whitelist which modules can be loaded
`module_allowed_unsafe(mod, filename : string) : bool`	No	Control whether `unsafe` blocks are permitted
`option_allowed(opt, from : string) : bool`	No	Whitelist which `options` directives are accepted
`annotation_allowed(ann, from : string) : bool`	No	Whitelist which annotations are accepted
`can_module_be_required(mod, fname : string; isPublic : bool) : bool`	No	Fine-grained control over `require` statements
`is_pod_in_scope_allowed(modName, fileName : string) : bool`	No	Control `var inscope` for POD types
`include_get(inc, from : string) : string`	No	Resolve `include` directives
`is_same_file_name(a, b : string) : bool`	No	Custom file path comparison
`dyn_modules_folder() : string`	No	Set dynamic module search folder

`module_info` is defined as `tuple<string; string; string> const` — the three fields are `(moduleName, fileName, importName)`.

Callbacks that are not exported fall back to permissive defaults (everything allowed). If `module_get` is missing, the project is treated as failed and all callbacks revert to defaults.

This tutorial demonstrates the five most impactful callbacks.

module_get — path resolution

When a `.das_project` is active, the built-in `daslib/` resolution logic is **completely bypassed** — `module_get` is the sole path resolver. It must handle `daslib/X` paths as well as relative module paths:

```
[export]
def module_get(req, from : string) : module_info {
    let rs <- split_by_chars(req, "./")
    let mod_name = rs[length(rs) - 1]
    if (length(rs) == 2 && rs[0] == "daslib") {
        return (mod_name, "{get_das_root()}/daslib/{mod_name}.das", "")
    }
    // Resolve relative to the requiring file
    var fr <- split_by_chars(from, "/")
    if (length(fr) > 0) { pop(fr) }
    for (se in rs) { push(fr, se) }
```

(continues on next page)

(continued from previous page)

```
    return (mod_name, join(fr, "/") + ".das", "")
}
```

module_allowed — module whitelist

Called for **every** module (native C++ and compiled `.das`) that is loaded during compilation. The `mod` parameter is the short module name — for example `"strings"`, `"fio"`, `"sandbox_helpers"` — not the `require` path.

The built-in core module `"$"` must always be whitelisted:

```
[export]
def module_allowed(mod, filename : string) : bool {
    if (mod == "$") { return true }
    if (mod == "math" || mod == "strings") { return true }
    if (mod == "strings_boost" || mod == "sandbox_helpers") { return true }
    return false
}
```

Returning `false` produces a `module not allowed` compilation error.

module_allowed_unsafe — forbid unsafe

Called once per compiled module. Returning `false` forbids all `unsafe` blocks in that module:

```
[export]
def module_allowed_unsafe(mod, filename : string) : bool {
    return false   // no module may use unsafe
}
```

option_allowed — restrict options

Called during parsing for each `options` declaration. Returning `false` rejects the option with an `invalid option` error:

```
[export]
def option_allowed(opt, from : string) : bool {
    if (opt == "gen2" || opt == "indenting") { return true }
    return false
}
```

annotation_allowed — restrict annotations

Called for each annotation used in the script:

```
[export]
def annotation_allowed(ann, from : string) : bool {
    if (ann == "export" || ann == "private") { return true }
    return false
}
```

The C host code

The tutorial embeds four inline daslang scripts as C string constants:

- **HELPER_MODULE** — a utility module (`sandbox_helpers`) with two simple functions (`clamp_value` and `greet`)
- **VALID_SCRIPT** — requires allowed modules, calls helper functions
- **VIOLATES_OPTION** — uses `options persistent_heap` (banned)
- **VIOLATES_UNSAFE** — uses an `unsafe` block (forbidden)
- **VIOLATES_MODULE** — requires `fio` (not whitelisted)

Setting up the sandbox

```
// 1. Load the .das_project
das_file_access * fa = das_fileaccess_make_project(project_path);

// 2. Pre-cache daslib (needed for module path resolution)
das_fileaccess_introduce_daslib(fa);

// 3. Introduce user files
das_fileaccess_introduce_file(fa, "sandbox_helpers.das", HELPER_MODULE, 0);
das_fileaccess_introduce_file(fa, "user_script.das", VALID_SCRIPT, 0);

// 4. Lock filesystem
das_fileaccess_lock(fa);

// 5. Compile - only cached files accessible, policy enforced
das_program * program = das_program_compile("user_script.das",
                                            fa, tout, module_group);
```

Reporting violations

When policy violations occur, the compiler reports them as errors. Iterate `das_program_get_error` and format with `das_error_report`:

```
int err_count = das_program_err_count(program);
for (int i = 0; i < err_count; i++) {
    das_error * error = das_program_get_error(program, i);
    char buf[1024];
    das_error_report(error, buf, sizeof(buf));
    printf("  error %d: %s\n", i, buf);
}
```

Building and running

Build with CMake:

```
cmake --build build --config Release --target integration_c_06
```

Run:

```
bin/Release/integration_c_06
```

Expected output (Part 1 — valid script):

```
=== Part 1: Valid script in sandbox ===

clamped: 100
Hello, sandbox!
split: [[ a; b; c]]
```

Expected output (Part 2 — banned option):

```
=== Part 2: Banned option ===

Compilation produced 1 error(s):
  error 0: option persistent_heap is not allowed here ...
```

Expected output (Part 3 — unsafe forbidden):

```
=== Part 3: Unsafe block forbidden ===

Compilation produced 1 error(s):
  error 0: unsafe function test ...
```

Expected output (Part 4 — banned module):

```
=== Part 4: Banned module ===

Compilation produced 1 error(s):
  error 0: module not allowed 'fio' ...
```

Expected output (Part 5 — CodeOfPolicies via C API):

```
=== Part 5: CodeOfPolicies via C API ===

Compilation with DAS_POLICY_NO_UNSAFE produced 1 error(s):
  error 0: unsafe function test ...
```

Part 5 uses `das_policies_make` and `das_policies_set_bool` to set `DAS_POLICY_NO_UNSAFE` directly from C, without a `.das_project` file. The error is identical to the one produced by the project-based sandbox in Part 3.

See also:

Full source: `06_sandbox.c`, `06_sandbox.das_project`

Previous tutorial: tutorial_integration_c_unaligned_advanced

Next tutorial: tutorial_integration_c_context_variables

type_mangling — complete type mangling reference

daScriptC.h API header: `include/daScript/daScriptC.h`

5.2.7 C Integration: Context Variables

This tutorial demonstrates how to **read, write, and enumerate** daslang global variables from a C host using the `daScriptC.h` API.

After a program is compiled and simulated, its global variables live in the context. The C API provides five functions to access them:

Function	Purpose
`das_context_find_variable(ctx, name)`	Look up a global by name; returns an index
`das_context_get_variable(ctx, idx)`	Get a raw `void*` to the variable storage
`das_context_get_total_variables(ctx)`	Total number of globals
`das_context_get_variable_name(ctx, i)`	Name of the i-th global
`das_context_get_variable_size(ctx, i)`	Size (in bytes) of the i-th global

The daslang script

The companion script declares four scalar globals:

```
options gen2

var score : int = 42
var speed : float = 3.14
var player_name : string = "Hero"
var alive : bool = true

[export]
def print_globals() {
    print("  score       = {score}\n")
    print("  speed       = {speed}\n")
    print("  player_name = {player_name}\n")
    print("  alive       = {alive}\n")
}
```

Reading globals

After simulation, look up a global by name and cast the returned pointer to the appropriate C type:

```
int idx = das_context_find_variable(ctx, "score");
if (idx >= 0) {
    int * p = (int *)das_context_get_variable(ctx, idx);
    printf("score = %d\n", *p);
}
```

String globals are stored as `char*` internally:

```
int idx = das_context_find_variable(ctx, "player_name");
if (idx >= 0) {
    char ** p = (char **)das_context_get_variable(ctx, idx);
    printf("player_name = %s\n", *p);
}
```

`bool` is stored as a single byte (not `int`):

```
char * p = (char *)das_context_get_variable(ctx, idx_alive);
printf("alive = %s\n", *p ? "true" : "false");
```

Writing globals

The pointer returned by `das_context_get_variable` is **read-write**. Modifications are immediately visible to daslang functions:

```
int * p = (int *)das_context_get_variable(ctx, idx_score);
*p = 9999;    // next call to a daslang function sees score == 9999
```

Enumerating all globals

```
int total = das_context_get_total_variables(ctx);
for (int i = 0; i < total; i++) {
    const char * name = das_context_get_variable_name(ctx, i);
    int sz = das_context_get_variable_size(ctx, i);
    printf("  [%d] %s (size=%d)\n", i, name, sz);
}
```

Build & run

Build:

```
cmake --build build --config Release --target integration_c_07
```

Run:

```
bin/Release/integration_c_07
```

Expected output:

```
=== Reading globals from C ===
score = 42
speed = 3.14
player_name = Hero
alive = true

=== Writing globals from C ===
C set score = 9999
C set speed = 99.5

=== Calling daslang to verify ===
  score       = 9999
  speed       = 99.5
  player_name = Hero
  alive       = true

=== All global variables ===
  [0] score (size=4)
  [1] speed (size=4)
  [2] player_name (size=8)
  [3] alive (size=1)
```

See also:

Full source: `07_context_variables.c`, `07_context_variables.das`

Previous tutorial: tutorial_integration_c_sandbox

Next tutorial: tutorial_integration_c_serialization

C++ equivalent: tutorial_integration_cpp_context_variables

daScriptC.h API header: `include/daScript/daScriptC.h`

5.2.8 C Integration: Serialization

This tutorial demonstrates how to **serialize** a compiled daslang program to a binary blob and **deserialize** it back, skipping recompilation on subsequent runs.

The workflow:

1. Compile a program normally with `das_program_compile`
2. Simulate it at least once (resolves function pointers)
3. Serialize to an in-memory buffer
4. Release the original program
5. Deserialize from the buffer — no parsing, no type inference
6. Simulate and run as usual

This is useful for faster startup, distributing pre-compiled scripts, and caching compiled programs in editors or build pipelines.

Serialization API

Three functions make up the serialization API:

Function	Purpose
das_program_serialize(prog, &data, &size)	Serialize to binary; returns opaque handle
das_program_deserialize(data, size)	Deserialize from raw bytes; returns program
das_serialized_data_release(blob)	Free the serialization buffer

Serializing

```
const void * blob_data = NULL;
int64_t blob_size = 0;
das_serialized_data * blob = das_program_serialize(program, &blob_data, &blob_size);

printf("Serialized size: %lld bytes\n", (long long)blob_size);

// Save blob_data/blob_size to file for persistent caching if desired.
```

The program must have been simulated at least once before serialization.

Deserializing

```
das_program * restored = das_program_deserialize(blob_data, blob_size);

// The blob can be released after deserialization - the program is independent.
das_serialized_data_release(blob);

// Simulate and run as usual
das_context * ctx = das_context_make(das_program_context_stack_size(restored));
das_program_simulate(restored, ctx, tout);

das_function * fn = das_context_find_function(ctx, "test");
das_context_eval_with_catch(ctx, fn, NULL);
```

Build & run

Build:

```
cmake --build build --config Release --target integration_c_08
```

Run:

```
bin/Release/integration_c_08
```

Expected output:

```
=== Stage 1: Compile from source ===
  Compiled successfully.
  Simulated successfully.
```

(continues on next page)

(continued from previous page)

```
=== Stage 2: Serialize ===
  Serialized size: 5022 bytes
  Original program released.

=== Stage 3: Deserialize ===
  Deserialized successfully.

=== Stage 4: Simulate and run ===
=== Serialization Tutorial ===
Hello, World!
sum_to(10)  = 55
sum_to(100) = 5050
```

See also:

Full source: `08_serialization.c`

This tutorial reuses the script from the C++ serialization tutorial: `14_serialization.das`

Previous tutorial: tutorial_integration_c_context_variables

Next tutorial: tutorial_integration_c_aot

C++ equivalent: tutorial_integration_cpp_serialization

daScriptC.h API header: `include/daScript/daScriptC.h`

5.2.9 C Integration: AOT

This tutorial demonstrates **Ahead-of-Time compilation** via the C API. AOT translates daslang functions into C++ source code at build time. When linked into the host executable and enabled via policies, `simulate()` replaces interpreter nodes with direct native calls — near-C++ performance.

AOT workflow

1. **Generate** C++ from the script: `daslang.exe utils/aot/main.das -- -aot script.das script.das.cpp`
2. **Compile** the generated `.cpp` into the host executable (CMake handles this)
3. **Set policies**: `DAS_POLICY_AOT = 1` before compilation
4. **Simulate**: the runtime links AOT functions automatically
5. **Verify**: `das_function_is_aot(fn)` returns 1

Using policies from C

The `das_policies` API lets you control `CodeOfPolicies` from C:

```
das_policies * pol = das_policies_make();
das_policies_set_bool(pol, DAS_POLICY_AOT, 1);
das_policies_set_bool(pol, DAS_POLICY_FAIL_ON_NO_AOT, 1);

das_program * program = das_program_compile_policies(
    script, fa, tout, libgrp, pol);
das_policies_release(pol);
```

Two enum types ensure type safety:

- `das_bool_policy` — boolean flags (`DAS_POLICY_AOT`, `DAS_POLICY_NO_UNSAFE`, `DAS_POLICY_NO_GLOBAL_VARIABLES`, etc.)
- `das_int_policy` — integer fields (`DAS_POLICY_STACK`, `DAS_POLICY_MAX_HEAP_ALLOCATED`, etc.)

Using the wrong enum type will produce a compiler warning.

Checking AOT status

After simulation, check whether a function was AOT-linked:

```
das_function * fn = das_context_find_function(ctx, "test");
printf("test() is AOT: %s\n", das_function_is_aot(fn) ? "yes" : "no");
```

Available boolean policies

Enum value	CodeOfPolicies field
`DAS_POLICY_AOT`	`aot` — enable AOT linking
`DAS_POLICY_NO_UNSAFE`	`no_unsafe` — forbid unsafe blocks
`DAS_POLICY_NO_GLOBAL_VARIABLES`	`no_global_variables` — forbid module-level var
`DAS_POLICY_NO_GLOBAL_HEAP`	`no_global_heap` — forbid heap globals
`DAS_POLICY_NO_INIT`	`no_init` — forbid [init] functions
`DAS_POLICY_FAIL_ON_NO_AOT`	`fail_on_no_aot` — missing AOT is error
`DAS_POLICY_THREADLOCK_CONTEXT`	`threadlock_context` — context mutex
`DAS_POLICY_INTERN_STRINGS`	`intern_strings` — string interning
`DAS_POLICY_PERSISTENT_HEAP`	`persistent_heap` — no GC between calls
`DAS_POLICY_MULTIPLE_CONTEXTS`	`multiple_contexts` — context safety
`DAS_POLICY_STRICT_SMART_POINTERS`	`strict_smart_pointers` — var inscope rules
`DAS_POLICY_RTTI`	`rtti` — extended RTTI
`DAS_POLICY_NO_OPTIMIZATIONS`	`no_optimizations` — disable all optimizations

Available integer policies

Enum value	CodeOfPolicies field
DAS_POLICY_STACK	stack — context stack size (bytes)
DAS_POLICY_MAX_HEAP_ALLOCATED	max_heap_allocated — max heap (0 = unlimited)
DAS_POLICY_MAX_STRING_HEAP_ALLOCATED	max_string_heap_allocated
DAS_POLICY_HEAP_SIZE_HINT	heap_size_hint — initial heap size
DAS_POLICY_STRING_HEAP_SIZE_HINT	string_heap_size_hint

Build & run

Build:

```
cmake --build build --config Release --target integration_c_09
```

Run:

```
bin/Release/integration_c_09
```

Expected output:

```
=== Interpreter mode ===
  test() is AOT: no
=== AOT Tutorial ===
fib(0) = 0
fib(1) = 1
...
fib(9) = 34
sin(pi/4) = 0.70710677
cos(pi/4) = 0.70710677
sqrt(2)   = 1.4142135

=== AOT mode (policies.aot = true) ===
  test() is AOT: yes
=== AOT Tutorial ===
fib(0) = 0
...
fib(9) = 34
```

See also:

Full source: 09_aot.c

This tutorial reuses the script from the C++ AOT tutorial: 13_aot.das

Previous tutorial: tutorial_integration_c_serialization

Next tutorial: tutorial_integration_c_threading

C++ equivalent: tutorial_integration_cpp_aot

daScriptC.h API header: include/daScript/daScriptC.h

5.2.10 C Integration: Threading

This tutorial demonstrates how to use daslang contexts and compilation pipelines across multiple threads from a pure C host application using the `daScriptC.h` API.

Topics covered:

- **Part A** — Running a compiled context on a worker thread
- **Part B** — Compiling a script from scratch on a worker thread
- `das_environment_get_bound()` / `das_environment_set_bound()` — thread-local environment binding
- `das_reuse_cache_guard_create()` / `das_reuse_cache_guard_release()` — thread-local free-list cache management
- `das_context_clone()` with `DAS_CONTEXT_CATEGORY_THREAD_CLONE`
- `DAS_POLICY_THREADLOCK_CONTEXT` — context mutex for threads

Prerequisites

- Tutorial 1 completed (tutorial_integration_c_hello_world) — basic compile → simulate → eval cycle.
- Familiarity with platform threading primitives (`_beginthreadex` on Windows, `pthread_create` on POSIX).

Why threading matters

daslang uses **thread-local storage** (TLS) for its environment object. Every thread that touches the daslang C API must have a valid environment bound. There are two common patterns:

1. **Share the environment** — the worker thread binds the same environment as the main thread via `das_environment_set_bound()`. Use this when the worker only *executes* and the main thread is idle.
2. **Independent environment** — the worker thread calls `das_initialize()` / `das_shutdown()` to create and destroy its own module registry and compilation pipeline.

New C API functions for threading

These functions were added to `daScriptC.h` specifically for multi-threaded host applications:

Function	Purpose
`das_environment_get_bound()`	Return the environment bound to the calling thread (TLS).
`das_environment_set_bound(env)`	Bind an environment on the calling thread.
`das_reuse_cache_guard_create()`	Create a thread-local free-list cache guard (call first on any worker thread).
`das_reuse_cache_guard_release(guard)`	Release the guard (call before thread exit).
`das_context_clone(ctx, category)`	Clone a context for use on another thread.
`DAS_CONTEXT_CATEGORY_THREAD_CLONE`	Category constant for thread-owned clones.

The daslang script

A simple script that computes the sum 0 + 1 + … + 99 = 4950:

```
options gen2

[export]
def compute() : int {
    var total = 0
    for (i in range(100)) {
        total += i
    }
    return total
}
```

Part A — Run on a worker thread

Compile and simulate on the main thread, then clone the context and run it on a worker thread.

```
// Compile with threadlock_context enabled
das_policies * pol = das_policies_make();
das_policies_set_bool(pol, DAS_POLICY_THREADLOCK_CONTEXT, 1);
das_program * prog = das_program_compile_policies(path, fac, tout, lib, pol);
das_policies_release(pol);

// Simulate on the main thread
das_context * ctx = das_context_make(das_program_context_stack_size(prog));
das_program_simulate(prog, ctx, tout);

// Clone the context for the worker thread
das_context * clone = das_context_clone(ctx,
                          DAS_CONTEXT_CATEGORY_THREAD_CLONE);

// Capture the environment
das_environment * env = das_environment_get_bound();

// On the worker thread:
das_environment_set_bound(env);
das_function * fn = das_context_find_function(clone, "compute");
vec4f res = das_context_eval_with_catch(clone, fn, NULL);
int result = das_argument_int(res);
```

Key points:

Function	Explanation
`das_environment_get_bound()`	Returns the current thread's TLS environment pointer. Must be captured **before** launching the worker.
`das_environment_set_bound(env)`	Binds the environment on the worker thread. Without this, any daslang C API call will crash.
`das_context_clone()`	Creates a new context that shares the simulated program. Release with `das_context_release()`.
`DAS_CONTEXT_CATEGORY_THREAD_CLONE`	Marks the clone as thread-owned for diagnostics and safety.
`DAS_POLICY_THREADLOCK_CONTEXT`	Compile-time policy that gives the context a mutex. Required when the context (or its clone) runs on a non-main thread.

Part B — Compile on a worker thread

Create a fully independent daslang environment on a new thread.

```
// On the worker thread:

// 1. Thread-local free-list cache - must be first.
das_reuse_cache_guard * guard = das_reuse_cache_guard_create();

// 2+3. Register modules + create environment in TLS.
das_initialize();

// 4. Standard compile → simulate → eval cycle.
das_policies * pol = das_policies_make();
das_policies_set_bool(pol, DAS_POLICY_THREADLOCK_CONTEXT, 1);
das_program * prog = das_program_compile_policies(path, fac, tout, lib, pol);
das_policies_release(pol);
das_context * ctx = das_context_make(das_program_context_stack_size(prog));
das_program_simulate(prog, ctx, tout);
das_function * fn = das_context_find_function(ctx, "compute");
vec4f res = das_context_eval_with_catch(ctx, fn, NULL);
int result = das_argument_int(res);

// Cleanup
das_context_release(ctx);
das_program_release(prog);
/* ... release fac, lib, tout ... */

// 5. Shut down this thread's module system.
das_shutdown();
das_reuse_cache_guard_release(guard);
```

Key points:

Function	Explanation
`das_reuse_cache_guard_create()`	Initializes per-thread free-list caches. Must be created **first** on any thread that uses daslang.
`das_initialize()`	Registers module factories and creates a new environment in TLS. Safe to call on multiple threads — module registration is idempotent.
`das_shutdown()`	Destroys this thread's modules and environment. Must be called before the thread exits.
`das_reuse_cache_guard_release()`	Tears down thread-local caches. Call **after** `das_shutdown()`.
`DAS_POLICY_THREADLOCK_CONTEXT`	Same policy as Part A — ensures the context has a mutex.

Choosing the right pattern

Criteria	Part A (clone + run)	Part B (compile on thread)
Compilation cost	Zero on worker (pre-compiled)	Full compilation on worker
Isolation	Shares environment with main	Fully independent
Concurrent compilation	Not safe (shared env)	Safe (separate env per thread)
Use case	Game threads running pre-compiled AI/logic	Build servers, parallel test runners

Portable threading

The tutorial uses `_beginthreadex` on Windows and `pthread_create` on POSIX, wrapped in a small helper. The `DAS_C_INTEGRATION_TUTORIAL` CMake macro already links `Threads::Threads`, so no extra setup is needed for pthreads.

Build & run

```
cmake --build build --config Release --target integration_c_10
bin/Release/integration_c_10
```

Expected output:

```
=== Part A: Run on a worker thread ===
  compute() on worker thread returned: 4950
  PASS

=== Part B: Compile on a worker thread ===
  compute() compiled + run on worker thread returned: 4950
  PASS
```

See also:

Full source: `10_threading.c`, `10_threading.das`

Previous tutorial: tutorial_integration_c_aot

C++ equivalent: tutorial_integration_cpp_threading

daScriptC.h API header: `include/daScript/daScriptC.h`

5.3 C++ Integration Tutorials

These tutorials show how to embed daslang in a C++ application using the native `daScript.h` API. Each tutorial comes with a `.cpp` source file and a companion `.das` script in `tutorials/integration/cpp/`.

5.3.1 C++ Integration: Hello World

This tutorial shows how to embed daslang in a C++ application using the native `daScript.h` API. By the end you will have a standalone program that compiles a `.das` script, finds a function, calls it, and prints `Hello from daslang!`.

Note: If you have already read the C integration series (starting at tutorial_integration_c_hello_world), you will notice that the C++ API is considerably more concise: there is no manual reference counting, strings are `std::string`, and smart pointers manage lifetimes automatically.

Prerequisites

- daslang built from source (`cmake --build build --config Release`). The build produces `libDaScript` which the tutorial links against.
- The `daScript.h` header — located in `include/daScript/daScript.h`. This is the main C++ header that pulls in the full API (type system, compilation, contexts, module registration, etc.).

The daslang file

Create a minimal script with a single exported function:

```
options gen2

[export]
def test() {
    print("Hello from daslang!\n")
}
```

The `[export]` annotation makes the function visible to the host application so that `Context::findFunction` can locate it by name.

The C++ program

The program follows the same lifecycle as the C version, but each step uses the native C++ API directly.

Step 1 — Initialize

```
#include "daScript/daScript.h"

using namespace das;

int main(int, char * []) {
    NEED_ALL_DEFAULT_MODULES;
    Module::Initialize();
```

`NEED_ALL_DEFAULT_MODULES` is a macro that ensures all built-in modules (math, strings, etc.) are linked into the executable. `Module::Initialize` activates them. Both must be called **once** before any compilation.

Step 2 — Set up compilation infrastructure

```
TextPrinter tout;
ModuleGroup dummyLibGroup;
auto fAccess = make_smart<FsFileAccess>();
```

Object	Purpose
`TextPrinter`	Sends compiler/runtime messages to **stdout**. For an in-memory buffer, use `TextWriter` instead.
`ModuleGroup`	Holds modules available during compilation.
`FsFileAccess`	Provides disk-based file I/O to the compiler. Wrapped in a `smart_ptr` — freed automatically.

Step 3 — Compile the script

```
auto program = compileDaScript(getDasRoot() + SCRIPT_NAME,
                               fAccess, tout, dummyLibGroup);
if (program->failed()) {
    for (auto & err : program->errors) {
        tout << reportError(err.at, err.what, err.extra,
                            err.fixme, err.cerr);
    }
    return;
}
```

`compileDaScript` reads the `.das` file, parses, type-checks, and returns a `ProgramPtr` (a smart pointer to `Program`). `getDasRoot()` returns the project root directory, so paths are always resolved correctly regardless of the working directory.

Errors are stored in `program->errors` and can be formatted with `reportError`.

Step 4 — Create a context and simulate

```
Context ctx(program->getContextStackSize());
if (!program->simulate(ctx, tout)) {
    // handle errors ...
}
```

A **Context** is the runtime environment — it owns the execution stack, global variables, and heap. Unlike the C API, the context is a stack-allocated object here; there is no `release` call.

Simulation resolves function pointers, initializes globals, and prepares everything for execution. Always check for errors after `simulate`.

Step 5 — Find and verify the function

```
auto fnTest = ctx.findFunction("test");
if (!fnTest) {
    tout << "function 'test' not found\n";
    return;
}
if (!verifyCall<void>(fnTest->debugInfo, dummyLibGroup)) {
    tout << "wrong signature\n";
    return;
}
```

`findFunction` returns a `SimFunction *` — valid for the lifetime of the context.

`verifyCall<ReturnType, ArgTypes...>` is a compile-time-safe signature check. It is slow (it walks debug info), so do it **once** during setup — not on every call in a hot loop.

Step 6 — Call the function

```
ctx.evalWithCatch(fnTest, nullptr);
if (auto ex = ctx.getException()) {
    tout << "exception: " << ex << "\n";
}
```

`evalWithCatch` runs the function inside a C++ try/catch so that a daslang `panic()` does not crash the host. The second argument is an array of `vec4f` arguments — `nullptr` here because `test` takes none.

Step 7 — Shut down

```
    Module::Shutdown();
    return 0;
}
```

`Module::Shutdown` frees all global state. No daslang API calls are allowed after it. Note that unlike the C API, there is no need to manually release the program, module group, or text printer — these are either stack-allocated or reference-counted smart pointers.

C++ vs. C API comparison

C API (daScriptC.h)	C++ API (daScript.h)
das_initialize()	NEED_ALL_DEFAULT_MODULES; Module::Initialize()
das_text_make_printer() / das_text_release()	TextPrinter tout; (stack)
das_fileaccess_make_default() / das_fileaccess_release()	make_smart<FsFileAccess>() (ref-counted)
das_program_compile() / das_program_release()	compileDaScript() (returns ProgramPtr)
das_context_make() / das_context_release()	Context ctx(stackSize); (stack)
das_context_find_function()	ctx.findFunction()
das_context_eval_with_catch()	ctx.evalWithCatch()
das_shutdown()	Module::Shutdown()

Key concepts

Smart pointers
: The C++ API uses smart_ptr<T> (daslang's own reference-counted smart pointer) and make_smart<T>() for heap-allocated objects. ProgramPtr returned by compileDaScript is one example.

Stack-allocated objects
: TextPrinter, ModuleGroup, and Context can all live on the stack. Their destructors handle cleanup automatically.

getDasRoot
: Returns the root of the daslang installation as a string. Use it to build paths to scripts so they resolve regardless of working directory.

verifyCall
: Template function that validates a SimFunction's signature against expected C++ types. Use it as a development-time safety net.

Building and running

The tutorial is built automatically by CMake as part of the daslang project:

```
cmake --build build --config Release --target integration_cpp_01
```

Run from the project root so that the script path resolves correctly:

```
bin\Release\integration_cpp_01.exe
```

Expected output:

```
Hello from daslang!
```

See also:

Full source: 01_hello_world.cpp, 01_hello_world.das

Next tutorial: tutorial_integration_cpp_calling_functions

C API equivalent: tutorial_integration_c_hello_world

5.3.2 C++ Integration: Calling Functions

This tutorial shows **two ways** to call daslang functions from C++:

- **Part A — Low-level:** manual `cast<T>::from` / `to` + `evalWithCatch`. Gives maximum control over the calling convention.
- **Part B — High-level:** `das_invoke_function` / `das_invoke_function_by_name`. Type-safe variadic templates that handle marshalling and AOT dispatch automatically. **Prefer this in production code.**

Note: All argument and return value marshalling goes through `vec4f`, the SIMD-aligned register type that daslang uses for its calling convention. Part A works with `vec4f` directly; Part B hides it behind templates.

Prerequisites

- Tutorial 01 completed (tutorial_integration_cpp_hello_world).
- Familiarity with the daslang lifecycle (init → compile → simulate → eval → shutdown).

The daslang file

The script exports several functions with different signatures:

```
options gen2

[export]
def add(a : int; b : int) : int {
    return a + b
}

[export]
def square(x : float) : float {
    return x * x
}

[export]
def greet(name : string) : string {
    return "Hello, {name}!"
}

struct Vec2 {
    x : float
    y : float
}

[export]
def make_vec2(x : float; y : float) : Vec2 {
    return Vec2(x = x, y = y)
}

[export]
def will_fail() {
```

(continues on next page)

(continued from previous page)

```
    panic("something went wrong!")
}
```

Part A — Low-level calling

Passing arguments with `cast<T>::from`

Every daslang function takes arguments as an array of `vec4f`. Use `cast<T>::from(value)` to pack a C++ value into a `vec4f` slot:

```
vec4f args[2];
args[0] = cast<int32_t>::from(17);   // int
args[1] = cast<int32_t>::from(25);   // int

vec4f ret = ctx.evalWithCatch(fnAdd, args);
```

The supported primitive types are:

C++ type	`cast<>` specialization
`int32_t`	`cast<int32_t>`
`uint32_t`	`cast<uint32_t>`
`int64_t`	`cast<int64_t>`
`float`	`cast<float>`
`double`	`cast<double>`
`bool`	`cast<bool>`
`char *`	`cast<char *>`
pointers	`cast<T *>`

Reading return values with `cast<T>::to`

The return value of `evalWithCatch` is also a `vec4f`. Extract it with `cast<T>::to(ret)`:

```
int32_t result = cast<int32_t>::to(ret);
```

For strings, the returned `char *` points into the context's heap and remains valid until the context is destroyed or garbage-collected:

```
const char * result = cast<char *>::to(ret);
```

Verifying function signatures

`verifyCall<ReturnType, ArgTypes...>` checks a `SimFunction`'s debug info against the expected C++ types:

```
if (!verifyCall<int32_t, int32_t, int32_t>(fnAdd->debugInfo, dummyLibGroup)) {
    tout << "'add' has wrong signature\n";
    return;
}
```

This is a **development-time safety net** — it walks debug info and is slow, so call it once during setup, never in a hot loop.

Functions returning structs (cmres)

When a daslang function returns a struct, the caller supplies a result buffer. Use the three-argument form of `evalWithCatch`:

```
// C++ struct layout must exactly match the daslang struct.
struct Vec2 { float x; float y; };

Vec2 v;
ctx.evalWithCatch(fnMakeVec2, args, &v);
```

The third argument (`void * cmres`) is where the result is written.

Handling exceptions

If the script calls `panic()`, `evalWithCatch` catches it. Check with `getException()`:

```
ctx.evalWithCatch(fnFail, nullptr);
if (auto ex = ctx.getException()) {
    printf("Caught: %s\n", ex);
}
```

Part B — High-level calling

`das_invoke_function` (from `daScript/simulate/aot.h`) is a family of variadic templates that handle argument marshalling, AOT dispatch, and return-value extraction in a single call. **This is the recommended way to call daslang functions from C++.**

Include the header:

```
#include "daScript/simulate/aot.h"
```

Obtaining a `Func` handle

The `Func` struct (`daScript/misc/arraytype.h`) is a lightweight wrapper around `SimFunction *`. Convert a found `SimFunction *` into a `Func`:

```
auto fnAdd = ctx.findFunction("add");
Func add_func(fnAdd);
```

`Func` is small and copyable — pass it by value.

Calling with invoke

For functions returning scalars or strings, use das_invoke_function<RetType>::invoke:

```
int32_t result = das_invoke_function<int32_t>::invoke(&ctx, nullptr,
    add_func, 17, 25);
// result == 42

float sq = das_invoke_function<float>::invoke(&ctx, nullptr,
    square_func, 3.5f);
// sq == 12.25

char * greeting = das_invoke_function<char *>::invoke(&ctx, nullptr,
    greet_func, "World");
// greeting == "Hello, World!"
```

The second argument (LineInfo *) can be nullptr in most cases. It is used for error reporting and stack traces.

Calling with invoke_cmres

When a daslang function returns a struct, use invoke_cmres instead:

```
Vec2 v = das_invoke_function<Vec2>::invoke_cmres(&ctx, nullptr,
    make_vec2_func, 1.5f, 2.5f);
// v.x == 1.5, v.y == 2.5
```

This allocates the result buffer on the stack and passes it as the caller-managed result pointer.

Calling by name

das_invoke_function_by_name looks up the function by name at call time. It validates that exactly one function matches and that it is not private:

```
int32_t result = das_invoke_function_by_name<int32_t>::invoke(
    &ctx, nullptr, "add", 100, 200);
// result == 300
```

This is convenient for one-off calls but slower than using a Func handle. For functions called repeatedly, cache the Func instead.

Part A vs Part B comparison

Aspect	Part A (cast + evalWithCatch)	Part B (das_invoke_function)
Header	daScript.h	simulate/aot.h
Argument marshalling	Manual vec4f[]	Automatic (variadic templates)
Return value extraction	Manual cast<T>::to	Automatic (template parameter)
Struct returns	evalWithCatch(f, a, &buf)	invoke_cmres(...)
AOT dispatch	No	Yes
Exception handling	getException() after call	Throws C++ exception
Best for	Learning / edge cases	Production code

Building and running

```
cmake --build build --config Release --target integration_cpp_02
bin\Release\integration_cpp_02.exe
```

Expected output:

```
=== Part A: Low-level (cast + evalWithCatch) ===
  add(17, 25) = 42
  square(3.5) = 12.25
  greet("World") = "Hello, World!"
int=42 float=3.14 string=test bool=true
  make_vec2(1.5, 2.5) = { x=1.5, y=2.5 }
  Caught expected exception: something went wrong!

=== Part B: High-level (das_invoke_function) ===
  add(17, 25) = 42
  square(3.5) = 12.25
  greet("World") = "Hello, World!"
int=42 float=3.14 string=test bool=true
  make_vec2(1.5, 2.5) = { x=1.5, y=2.5 }
  add(100, 200) by name = 300
```

See also:

Full source: `02_calling_functions.cpp`, `02_calling_functions.das`

Previous tutorial: tutorial_integration_cpp_hello_world

Next tutorial: tutorial_integration_cpp_binding_functions

C API equivalent: tutorial_integration_c_calling_functions

5.3.3 C++ Integration: Binding Functions

This tutorial shows how to expose C++ functions to daslang scripts by creating a **custom module**. Topics covered:

- `addExtern` with `DAS_BIND_FUN` — binding free C++ functions
- `SideEffects` flags — telling the optimizer what a function does
- `addConstant` — exposing compile-time constants
- Context-aware functions — receiving `Context *` automatically

Prerequisites

- Tutorial 02 completed (tutorial_integration_cpp_calling_functions).
- Understanding of `cast<T>` and the daslang calling convention.

Creating a module

A module is a class derived from `Module`. Its constructor registers types, functions, and constants that scripts can use via `require`:

```
class Module_Tutorial03 : public Module {
public:
    Module_Tutorial03() : Module("tutorial_03_cpp") {
        ModuleLibrary lib(this);
        lib.addBuiltInModule();
        // ... register functions and constants here ...
    }
};

REGISTER_MODULE(Module_Tutorial03);
```

`REGISTER_MODULE` makes the module available via `NEED_MODULE` in the host program.

Binding constants

`addConstant` exposes a C++ value as a compile-time constant:

```
addConstant(*this, "PI",    3.14159265358979323846f);
addConstant(*this, "SQRT2", sqrtf(2.0f));
```

In the script:

```
print("PI = {PI}\n")       // 3.1415927
print("SQRT2 = {SQRT2}\n") // 1.4142135
```

Binding functions with `addExtern`

`addExtern` is the primary way to expose a C++ function to daslang. The `DAS_BIND_FUN` macro generates the template machinery needed for automatic argument marshalling:

```
float xmadd(float a, float b, float c, float d) {
    return a * b + c * d;
}

addExtern<DAS_BIND_FUN(xmadd)>(*this, lib, "xmadd",
    SideEffects::none, "xmadd");
```

Parameters:

1. `DAS_BIND_FUN(xmadd)` — the C++ function to bind
2. `*this` — the module being populated
3. `lib` — the module library (for type resolution)
4. `"xmadd"` — the name visible in daslang
5. `SideEffects::none` — optimizer hint (see below)
6. `"xmadd"` — C++ function name for AOT (used in generated code)

`SideEffects` flags

The `SideEffects` enum tells the optimizer what observable effects a function has. This controls whether calls can be eliminated, reordered, or folded:

Flag	Meaning
`none`	Pure function — no side effects. Safe to eliminate if result is unused.
`modifyExternal`	Modifies external state (stdout, files, hardware, etc.)
`modifyArgument`	Modifies one or more of its arguments (passed by reference).
`accessGlobal`	Reads global/shared mutable state.
`invoke`	Calls a daslang function or lambda.
`worstDefault`	Combines `modifyArgument | modifyExternal`. Use when unsure.

Example — a function that prints to stdout needs `modifyExternal`:

```
void greet(const char * name) {
    printf("Hello from C++, %s!\n", name);
}

addExtern<DAS_BIND_FUN(greet)>(*this, lib, "greet",
    SideEffects::modifyExternal, "greet");
```

Functions that modify arguments

If a function takes a reference and modifies it, use `SideEffects::modifyArgument`:

```
void double_it(int32_t & value) {
    value *= 2;
}

addExtern<DAS_BIND_FUN(double_it)>(*this, lib, "double_it",
    SideEffects::modifyArgument, "double_it");
```

In the script:

```
var val = 21
double_it(val)
print("{val}\n")    // 42
```

Context-aware functions

If a C++ function takes `Context *` as its first parameter (or `LineInfoArg *` as its last), daslang injects them automatically — the script caller does **not** see these parameters:

```
void print_stack_info(Context * ctx) {
    printf("Stack size: %d bytes\n", ctx->stack.size());
}

addExtern<DAS_BIND_FUN(print_stack_info)>(*this, lib, "print_stack_info",
    SideEffects::modifyExternal, "print_stack_info");
```

In the script the function takes zero arguments:

```
print_stack_info()   // prints "Stack size: 16384 bytes"
```

Activating the module in the host

The host program must request the module before `Module::Initialize()`:

```
int main(int, char * []) {
    NEED_ALL_DEFAULT_MODULES;
    NEED_MODULE(Module_Tutorial03);
    Module::Initialize();
    // ... compile and run scripts ...
    Module::Shutdown();
    return 0;
}
```

The script uses `require` to access the module:

```
require tutorial_03_cpp
```

Building and running

```
cmake --build build --config Release --target integration_cpp_03
bin\Release\integration_cpp_03.exe
```

Expected output:

```
PI    = 3.1415927
SQRT2 = 1.4142135
xmadd(SQRT2, SQRT2, 1.0, 1.0) = 3
factorial(10) = 3628800
Hello from C++, daslang!
double_it(21) = 42
Context stack size: 16384 bytes
```

See also:

Full source: `03_binding_functions.cpp`, `03_binding_functions.das`

Previous tutorial: tutorial_integration_cpp_calling_functions

Next tutorial: tutorial_integration_cpp_binding_types

C API equivalent: tutorial_integration_c_binding_types (the C tutorials combine type and function binding in a single tutorial)

5.3.4 C++ Integration: Binding Types

This tutorial shows how to expose C++ structs to daslang so that scripts can create instances, access fields, and pass them to/from C++ functions. Topics covered:

- `MAKE_TYPE_FACTORY` — registering a C++ type with the daslang type system
- `ManagedStructureAnnotation` — describing struct fields
- `addAnnotation` — plugging type metadata into a module
- `SimNode_ExtFuncCallAndCopyOrMove` — returning bound types by value
- Factory functions — creating instances without `unsafe`

Prerequisites

- Tutorial 03 completed (tutorial_integration_cpp_binding_functions).
- Familiarity with `addExtern` and `SideEffects`.

Defining the C++ types

We define three simple structs. They are intentionally **POD** (no default member initializers, no virtual functions) so that the daslang type system sees them as plain data:

```
struct Vec2 {
    float x;
    float y;
};

struct Color {
    uint8_t r;
    uint8_t g;
    uint8_t b;
    uint8_t a;
};

struct Rect {
    Vec2  pos;
    Vec2  size;
};
```

MAKE_TYPE_FACTORY

Before daslang can work with a C++ type, you must declare a **type factory** at file scope. `MAKE_TYPE_FACTORY` creates two things: `typeFactory<CppType>` (so `addExtern` resolves the type in function signatures) and `typeName<CppType>` (the type's display name):

```
MAKE_TYPE_FACTORY(Vec2,  Vec2);
MAKE_TYPE_FACTORY(Color, Color);
MAKE_TYPE_FACTORY(Rect,  Rect);
```

The first argument is the daslang-visible name; the second is the C++ type. They can differ when the C++ name lives in a namespace — e.g. `MAKE_TYPE_FACTORY(Vec2, math::Vec2)`.

ManagedStructureAnnotation

An **annotation** describes a type's layout to daslang. For structs, derive from `ManagedStructureAnnotation<T>` and call `addField` for each member:

```
struct Vec2Annotation : ManagedStructureAnnotation<Vec2, false> {
    Vec2Annotation(ModuleLibrary & ml)
        : ManagedStructureAnnotation("Vec2", ml)
    {
        addField<DAS_BIND_MANAGED_FIELD(x)>("x", "x");
        addField<DAS_BIND_MANAGED_FIELD(y)>("y", "y");
    }
};
```

The template parameters are `<CppType, canNew, canDelete>`. Passing `false` for `canNew` prevents scripts from calling `new Vec2()` directly (we provide factory functions instead).

`DAS_BIND_MANAGED_FIELD(member)` resolves the offset and type of a struct member at compile time. The two string arguments are the daslang field name and the C++ field name (used for AOT).

Nested bound types work naturally — `Rect` has `Vec2` fields:

```
struct RectAnnotation : ManagedStructureAnnotation<Rect, false> {
    RectAnnotation(ModuleLibrary & ml)
        : ManagedStructureAnnotation("Rect", ml)
    {
        addField<DAS_BIND_MANAGED_FIELD(pos)>("pos",   "pos");
        addField<DAS_BIND_MANAGED_FIELD(size)>("size", "size");
    }
};
```

Registering annotations in the module

Call `addAnnotation` in the module constructor. **Order matters** — if type B contains type A as a field, register A first:

```
addAnnotation(make_smart<Vec2Annotation>(lib));   // Vec2 first
addAnnotation(make_smart<ColorAnnotation>(lib));
addAnnotation(make_smart<RectAnnotation>(lib));   // Rect uses Vec2
```

Returning bound types by value

When a C++ function returns a bound struct by value, `addExtern` needs the `SimNode_ExtFuncCallAndCopyOrMove` sim-node so that the return value is properly copied into daslang's stack:

```
Vec2 vec2_add(const Vec2 & a, const Vec2 & b) {
    return { a.x + b.x, a.y + b.y };
}

addExtern<DAS_BIND_FUN(vec2_add), SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "vec2_add",
    SideEffects::none, "vec2_add")
        ->args({"a", "b"});
```

Functions that return scalars (`float`, `bool`, etc.) or take bound types by `const &` do not need this — the default sim-node works:

```
float vec2_length(const Vec2 & v) {
    return sqrtf(v.x * v.x + v.y * v.y);
}

addExtern<DAS_BIND_FUN(vec2_length)>(*this, lib, "vec2_length",
    SideEffects::none, "vec2_length")
        ->args({"v"});
```

Factory functions

Types bound via `ManagedStructureAnnotation` are **handled** (reference) types. Creating a mutable local variable of such a type requires an `unsafe` block. To give scripts a safe and ergonomic API, provide **factory functions** that return the type by value:

```
Vec2 make_vec2(float x, float y) {
    Vec2 v;  v.x = x;  v.y = y;
    return v;
}

addExtern<DAS_BIND_FUN(make_vec2), SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "make_vec2",
    SideEffects::none, "make_vec2")
        ->args({"x", "y"});
```

Using bound types in daslang

```
require tutorial_04_cpp

[export]
def test() {
    // Immutable locals - no `unsafe` needed
    let a = make_vec2(3.0, 4.0)
    print("length(a) = {vec2_length(a)}\n")

    let c = vec2_add(a, make_vec2(1.0, 2.0))
    print("a + b = ({c.x}, {c.y})\n")

    // Mutable local requires `unsafe`
    unsafe {
        var d = make_vec2(3.0, 4.0)
        vec2_normalize(d)
        print("normalize = ({d.x}, {d.y})\n")
    }

    // Nested types and field access
    let r = make_rect(10.0, 20.0, 100.0, 50.0)
    print("rect area = {rect_area(r)}\n")
```

Immutable locals created via `let` from factory functions work without `unsafe`. Use `unsafe { var ... }` only when the variable must be mutated (e.g. passed to a `modifyArgument` function).

Building and running

```
cmake --build build --config Release --target integration_cpp_04
bin\Release\integration_cpp_04.exe
```

Expected output:

```
a = (3, 4)
length(a) = 5
a + b = (4, 6)
a * 2 = (6, 8)
dot(a, b) = 11
normalize(3,4) = (0.6, 0.8)
color = (255, 128, 0, 255)
rect area = 5000
contains(50,30) = true
contains(200,200) = false
```

See also:

Full source: `04_binding_types.cpp`, `04_binding_types.das`

Previous tutorial: tutorial_integration_cpp_binding_functions

Next tutorial: tutorial_integration_cpp_binding_enums

5.3.5 C++ Integration: Binding Enumerations

This tutorial shows how to expose C++ `enum` and `enum class` types to daslang. Topics covered:

- `DAS_BASE_BIND_ENUM` — the macro approach to enum binding
- `DAS_BIND_ENUM_CAST` — cast specialization (when needed)
- `addEnumeration` — registering enums in a module
- Manual `Enumeration` construction — an alternative to macros
- Using bound enums in `addExtern` functions

Prerequisites

- Tutorial 04 completed (tutorial_integration_cpp_binding_types).
- Comfort with `MAKE_TYPE_FACTORY` and `addExtern`.

Defining the C++ enums

Both scoped (`enum class`) and unscoped (`enum`) enums can be bound. This tutorial uses scoped enums — the modern C++ style:

```
enum class Direction : int {
    North = 0,
    East  = 1,
    South = 2,
    West  = 3
};

enum class Severity : int {
    Debug   = 0,
    Info    = 1,
    Warning = 2,
    Error   = 3
};
```

DAS_BASE_BIND_ENUM

The primary macro for binding enums. It creates:

- A class `Enumeration<DasName>` that describes the enum's values
- A `typeFactory<CppEnum>` so that `addExtern` can resolve the type

```
DAS_BASE_BIND_ENUM(Direction, Direction,
    North,
    East,
    South,
    West
)

DAS_BASE_BIND_ENUM(Severity, Severity,
    Debug,
    Info,
    Warning,
    Error
)
```

The first argument is the C++ enum type, the second is the daslang name, and the remaining arguments are the enum values. The generated class names follow the pattern `Enumeration<DasName>` — e.g. `EnumerationDirection`.

Note: Place `DAS_BASE_BIND_ENUM` macros **before** `using namespace das`. The macros define names inside `namespace das` that can collide with the global enum names if both namespaces are active.

For unscoped (C-style) enums, use `DAS_BASE_BIND_ENUM_98` instead.

DAS_BIND_ENUM_CAST

This macro creates a `cast<>` specialization that lets enum values cross the C++ / daslang boundary. In many cases the SFINAE default in the engine already handles this, but you may need it for more complex enum types:

```
DAS_BIND_ENUM_CAST(Direction)
```

Registering enums in the module

In the module constructor, call `addEnumeration` with the generated class:

```
addEnumeration(make_smart<EnumerationDirection>());
addEnumeration(make_smart<EnumerationSeverity>());
```

Manual enum construction

When the macros don't fit (e.g. you need to rename values, skip some, or the enum lives in a deeply nested namespace), you can construct an `Enumeration` object by hand:

```
auto pEnum = make_smart<Enumeration>("Severity");
pEnum->cppName  = "Severity";
pEnum->external = true;
pEnum->baseType = Type::tInt;
pEnum->addIEx("Debug",   "Severity::Debug",   0, LineInfo());
pEnum->addIEx("Info",    "Severity::Info",    1, LineInfo());
pEnum->addIEx("Warning", "Severity::Warning", 2, LineInfo());
pEnum->addIEx("Error",   "Severity::Error",   3, LineInfo());
addEnumeration(pEnum);
```

You still need `DAS_BASE_BIND_ENUM` (or at least `DAS_BIND_ENUM_CAST`) for the `typeFactory<>` so that `addExtern` can match the type.

Binding functions that use enums

Once the enum type is registered, `addExtern` handles enum parameters and return values automatically — no special treatment is required:

```
Direction opposite_direction(Direction d) {
    switch (d) {
        case Direction::North: return Direction::South;
        case Direction::South: return Direction::North;
        case Direction::East:  return Direction::West;
        case Direction::West:  return Direction::East;
        default:               return d;
    }
}

addExtern<DAS_BIND_FUN(opposite_direction)>(*this, lib, "opposite_direction",
    SideEffects::none, "opposite_direction")
        ->args({"d"});
```

Using bound enums in daslang

Enum values are accessed with dot syntax — `EnumName.Value`:

```
require tutorial_05_cpp

[export]
def test() {
    let dir = Direction.North
    print("dir = {direction_name(dir)}\n")
    print("opposite = {direction_name(opposite_direction(dir))}\n")

    // Enum comparison
    let east = Direction.East
    let west = Direction.West
    print("opposite(East) == West? {opposite_direction(east) == west}\n")

    // Passing enums to C++ functions
    log_message(Severity.Warning, "disk space low")

    // Boolean result from enum logic
    print("Warning is severe? {is_severe(Severity.Warning)}\n")
```

Name collision warning

The daslang engine defines some enum names internally (e.g. `das::LogLevel` in `string_writer.h`). If your C++ enum has the same name as an engine-internal enum **and** you use `using namespace das`, you will get ambiguous symbol errors. Rename your enum to avoid the collision.

Building and running

```
cmake --build build --config Release --target integration_cpp_05
bin\Release\integration_cpp_05.exe
```

Expected output:

```
=== Direction enum ===
dir = North
opposite of North = South
Rotating CW: North East South West
East == West? false
opposite(East) == West? true

=== Severity enum ===
[DEBUG] system starting up
[INFO] ready to serve
[WARN] disk space low
[ERROR] connection lost
Debug is severe?   false
Warning is severe? true
Error is severe?   true
```

See also:

Full source: `05_binding_enums.cpp`, `05_binding_enums.das`

Previous tutorial: tutorial_integration_cpp_binding_types

Next tutorial: tutorial_integration_cpp_interop

5.3.6 C++ Integration: Low-Level Interop

This tutorial covers `addInterop` — the low-level alternative to `addExtern` for binding C++ functions to daslang. Topics covered:

- `addInterop` vs `addExtern` — when to use each
- Accepting "any type" arguments (`vec4f` template parameter)
- Runtime `TypeInfo` inspection via `call->types[i]`
- Accessing call-site debug info via `call->debugInfo`
- The `TypeInfo` union: `structType` / `enumType` / `annotation_or_name`

Prerequisites

- Tutorial 05 completed (tutorial_integration_cpp_binding_enums).
- Familiarity with `addExtern` and `ManagedStructureAnnotation`.

When to use `addInterop`

`addExtern` handles most function binding needs. Use `addInterop` when you need capabilities that `addExtern` cannot provide:

- **"Any type" arguments** — accept values of any daslang type and inspect them at runtime
- **TypeInfo access** — inspect type metadata like field names, enum values, struct layout
- **Call-site debug info** — know the source file and line number of the caller

The built-in `sprint`, `hash`, and `write` functions all use `addInterop` internally.

The `InteropFunction` signature

Every interop function must match this signature:

```
vec4f my_function(Context & context,
                  SimNode_CallBase * call,
                  vec4f * args);
```

- `context` — the script execution context
- `call` — the call node, provides `call->types[i]` (TypeInfo) and `call->debugInfo` (source location)
- `args` — the raw simulation-level arguments as `vec4f` values

Inspecting TypeInfo

`call->types[i]` returns a `TypeInfo *` for the *i*-th argument. The `type` field tells you what kind of type was passed:

```
vec4f describe_type(Context & context,
                    SimNode_CallBase * call,
                    vec4f * args) {
    TypeInfo * ti = call->types[0];
    TextWriter tw;

    if (ti->type == Type::tHandle) {
        auto ann = ti->getAnnotation();
        tw << "type = handle, name = " << ann->name;
    } else if (ti->type == Type::tStructure && ti->structType) {
        tw << "type = struct, name = " << ti->structType->name;
        tw << ", fields = " << ti->structType->count;
    } else {
        tw << "type = " << das_to_string(ti->type);
        tw << ", size = " << getTypeSize(ti);
    }

    auto result = context.allocateString(tw.str(), &call->debugInfo);
    return cast<char *>::from(result);
}
```

The TypeInfo union

`TypeInfo` contains a union of three pointers:

```
union {
    StructInfo *             structType;        // tStructure
    EnumInfo *               enumType;           // tEnumeration
    mutable TypeAnnotation * annotation_or_name; // tHandle
};
```

> **Warning:** Accessing the wrong union member is **undefined behavior**. Always check `ti->type` before accessing `structType`, `enumType`, or `annotation_or_name`.

For handled types (`type == tHandle`), use `ti->getAnnotation()` to safely resolve the annotation — it handles tagged-pointer resolution automatically. `das_to_string(Type::tHandle)` returns an empty string; use `ti->getAnnotation()->name` for the type name.

Registering interop functions

Use `addInterop<FuncPtr, ReturnType, ArgTypes...>`:

```
addInterop<describe_type, char *, vec4f>(*this, lib, "describe_type",
    SideEffects::none, "describe_type")
        ->arg("value");
```

When an `ArgType` is `vec4f`, it means "any daslang type." Concrete types like `int32_t` or `const char *` are also valid and work like `addExtern`.

Accessing call-site debug info

`call->debugInfo` provides the source location of the call:

```
vec4f call_site_info(Context & context,
                     SimNode_CallBase * call,
                     vec4f *) {
    TextWriter tw;
    if (call->debugInfo.fileInfo) {
        tw << call->debugInfo.fileInfo->name
           << ":" << call->debugInfo.line;
    }
    auto result = context.allocateString(tw.str(), &call->debugInfo);
    return cast<char *>::from(result);
}
```

Using interop functions from daslang

The interop functions look like regular functions in daslang — the "any type" argument accepts any value:

```
options gen2
require tutorial_06_cpp

struct MyPoint {
    x : float
    y : float
    tag : string
}

[export]
def test() {
    // Primitives and handled types
    print("{describe_type(42)}\n")
    print("{describe_type(true)}\n")
    let p = make_particle(1.0, 2.0, 0.5, -0.3)
    print("{describe_type(p)}\n")

    // Pure daslang struct - has StructInfo with field metadata
    let pt = MyPoint(x = 10.0, y = 20.0, tag = "origin")
    print("{describe_type(pt)}\n")
    print("fields: {struct_field_names(pt)}\n")
```

(continues on next page)

(continued from previous page)

```
    // Call-site info reports this script's file and line
    print("called from: {call_site_info()}\n")
```

Building and running

```
cmake --build build --config Release --target integration_cpp_06
bin\Release\integration_cpp_06.exe
```

Expected output:

```
=== describe_type ===
type = int, size = 4
type = float, size = 4
type = string, size = 8
type = bool, size = 1
type = handle, name = Particle, size = 16
type = struct, name = MyPoint, fields = 3, size = 16

=== debug_print ===
int:    42
float:  3.1400001f
string: "hello"
bool:   true

=== any_hash ===
hash(42)      = 0x...
hash("hello") = 0x...
hash(42) again = 0x...
same hash? true

=== struct_field_names ===
MyPoint fields: x, y, tag

=== call_site_info ===
called from: .../06_interop.das:57
```

See also:

Full source: `06_interop.cpp`, `06_interop.das`

Previous tutorial: tutorial_integration_cpp_binding_enums

Next tutorial: tutorial_integration_cpp_callbacks

5.3.7 C++ Integration: Callbacks

This tutorial shows how C++ code can receive and invoke daslang closures. Topics covered:

- `TBlock<Ret, Args...>` — typed block parameters (stack-bound closures)
- `das_invoke<Ret>::invoke()` — invoking a block from C++
- `Func` — function pointer parameters
- `das_invoke_function<Ret>::invoke()` — invoking a function pointer
- Lambda overview — `TLambda<>`, `das_invoke_lambda<>`
- Practical patterns: iteration, reduction

Prerequisites

- Tutorial 06 completed (tutorial_integration_cpp_interop).
- Familiarity with `addExtern` and `SideEffects`.

Blocks — `TBlock<Ret, Args...>`

Blocks are stack-bound closures — the primary callback mechanism in daslang. They are only valid **during the C++ call** that receives them. Never store a block for later use.

The template `TBlock<ReturnType, ArgType1, ArgType2, ...>` provides type safety: daslang's compiler checks the argument types at compile time.

```
void with_values(int32_t a, int32_t b,
                 const TBlock<void, int32_t, int32_t> & blk,
                 Context * context, LineInfoArg * at) {
    das_invoke<void>::invoke(context, at, blk, a, b);
}
```

A predicate block returning `bool`:

```
int32_t count_matching(const TArray<int32_t> & arr,
                       const TBlock<bool, int32_t> & pred,
                       Context * context, LineInfoArg * at) {
    int32_t count = 0;
    for (uint32_t i = 0; i < arr.size; ++i) {
        if (das_invoke<bool>::invoke(context, at, pred, arr[i])) {
            count++;
        }
    }
    return count;
}
```

Registration uses `SideEffects::invoke` because the function invokes script code:

```
addExtern<DAS_BIND_FUN(count_matching)>(*this, lib, "count_matching",
    SideEffects::invoke, "count_matching")
        ->args({"arr", "pred", "context", "at"});
```

Function pointers — Func

Func is a reference to a daslang-side function. Unlike blocks, function pointers can be stored and invoked multiple times (within the same context). In daslang, @@function_name creates a Func value.

```
void call_function_twice(Func fn, int32_t value,
                         Context * context, LineInfoArg * at) {
    das_invoke_function<void>::invoke(context, at, fn, value);
    das_invoke_function<void>::invoke(context, at, fn, value * 2);
}
```

Lambdas — Lambda / TLambda<>

Lambdas are heap-allocated closures that can capture variables. TLambda<Ret, Args...> is the typed variant (analogous to TBlock and TFunc). Invoke with das_invoke_lambda<Ret>::invoke().

Note: The untyped Lambda maps to lambda<> in daslang and will not match typed lambdas like lambda<(x:int):int>. Use TLambda<Ret, Args...> for type-safe lambda acceptance.

Practical pattern: C++ iterates, script processes

A common embedding pattern is having C++ own the iteration/generation logic while the script provides the processing callback:

```
void for_each_fibonacci(int32_t count,
                        const TBlock<void, int32_t, int32_t> & blk,
                        Context * context, LineInfoArg * at) {
    int32_t a = 0, b = 1;
    for (int32_t i = 0; i < count; ++i) {
        das_invoke<void>::invoke(context, at, blk, i, a);
        int32_t next = a + b;
        a = b;
        b = next;
    }
}
```

And the accumulator/reduce pattern:

```
int32_t reduce_range(int32_t from, int32_t to, int32_t init,
                     const TBlock<int32_t, int32_t, int32_t> & blk,
                     Context * context, LineInfoArg * at) {
    int32_t acc = init;
    for (int32_t i = from; i < to; ++i) {
        acc = das_invoke<int32_t>::invoke(context, at, blk, acc, i);
    }
    return acc;
}
```

Calling from daslang

Blocks use the <| pipe syntax with $() lambda prefix. Function pointers use @@function_name:

```
options gen2
require tutorial_07_cpp

def print_value(x : int) {
    print("fn({x})\n")
}

[export]
def test() {
    // Block callback
    with_values(10, 20) $(a, b) {
        print("a + b = {a + b}\n")
    }

    // Function pointer
    call_function_twice(@@print_value, 5)

    // Fibonacci iteration
    for_each_fibonacci(5) $(index, value) {
        print("fib({index}) = {value}\n")
    }

    // Reduce - sum 1..10
    let total = reduce_range(1, 11, 0) $(acc, i) {
        return acc + i
    }
    print("sum = {total}\n")

    // Reduce - factorial 10
    let fact = reduce_range(1, 11, 1) $(acc, i) {
        return acc * i
    }
    print("10! = {fact}\n")
```

Building and running

```
cmake --build build --config Release --target integration_cpp_07
bin\Release\integration_cpp_07.exe
```

Expected output:

```
=== Block: with_values ===
  a + b = 30

=== Block: count_matching ===
  even numbers: 5

=== Function pointer: call_function_twice ===
```

(continues on next page)

(continued from previous page)

```
  fn(5)
  fn(10)

=== Practical: for_each_fibonacci ===
  fib(0) = 0
  fib(1) = 1
  fib(2) = 1
  fib(3) = 2
  fib(4) = 3
  fib(5) = 5
  fib(6) = 8
  fib(7) = 13

=== Practical: reduce_range (sum 1..10) ===
  sum = 55

=== Practical: reduce_range (factorial 10) ===
  10! = 3628800
```

See also:

Full source: `07_callbacks.cpp`, `07_callbacks.das`

Previous tutorial: tutorial_integration_cpp_interop

Next tutorial: tutorial_integration_cpp_methods

5.3.8 C++ Integration: Binding Methods

This tutorial shows how to expose C++ member functions to daslang. Topics covered:

- `DAS_CALL_MEMBER` — wrapping a member function as a static callable
- `DAS_CALL_METHOD` — using the wrapper as an `addExtern` template argument
- `DAS_CALL_MEMBER_CPP` — providing the AOT-compatible name
- Const vs non-const methods and `SideEffects`
- Pipe syntax for method-like calls in daslang

Prerequisites

- Tutorial 07 completed (tutorial_integration_cpp_callbacks).
- Familiarity with `addExtern`, `ManagedStructureAnnotation`, and `MAKE_TYPE_FACTORY`.

How methods work in daslang

daslang does not have member functions. "Methods" are free functions whose first parameter is the struct instance (`self`). Pipe syntax (`obj |> method()`) provides method-call ergonomics.

DAS_CALL_MEMBER

This macro generates a static wrapper function that forwards to the C++ member function. Create a `using` alias for each method:

```
// Step 1: Create wrapper aliases
using method_increment = DAS_CALL_MEMBER(Counter::increment);
using method_get       = DAS_CALL_MEMBER(Counter::get);
using method_add       = DAS_CALL_MEMBER(Counter::add);
```

The wrapper's `invoke` method has the signature `invoke(Counter &, args...)` — the first argument is the object.

DAS_CALL_METHOD and DAS_CALL_MEMBER_CPP

Register the wrapper with `addExtern`:

```
// Step 2: Register methods
addExtern<DAS_CALL_METHOD(method_increment)>(*this, lib, "increment",
    SideEffects::modifyArgument,
    DAS_CALL_MEMBER_CPP(Counter::increment))
        ->args({"self"});

addExtern<DAS_CALL_METHOD(method_add)>(*this, lib, "add",
    SideEffects::modifyArgument,
    DAS_CALL_MEMBER_CPP(Counter::add))
        ->args({"self", "amount"});

addExtern<DAS_CALL_METHOD(method_get)>(*this, lib, "get",
    SideEffects::none,
    DAS_CALL_MEMBER_CPP(Counter::get))
        ->args({"self"});
```

Key points:

- **Non-const methods** (`increment`, `add`, `reset`) use `SideEffects::modifyArgument` because they mutate the object
- **Const methods** (`get`, `isZero`) use `SideEffects::none`
- `DAS_CALL_MEMBER_CPP(Counter::method)` provides the mangled name string for AOT compilation

Factory functions

Since handled types require `unsafe` for mutable local variables, provide factory functions that return by value so scripts can use `let`:

```
Counter make_counter(int32_t initial, int32_t step) {
    Counter c;
    c.value = initial;
    c.step = step;
    return c;
}

addExtern<DAS_BIND_FUN(make_counter),
          SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "make_counter",
    SideEffects::none, "make_counter")
        ->args({"initial", "step"});
```

Calling methods from daslang

The bound methods are called as free functions. Because handled types need mutable access under `unsafe`, the pattern is:

```
options gen2
require tutorial_08_cpp

[export]
def test() {
    unsafe {
        var c = make_counter(0, 1)
        increment(c)
        increment(c)
        add(c, 10)
        print("value = {get(c)}\n")  // 12
    }

    // Pipe syntax - idiomatic daslang
    unsafe {
        var c = make_counter(100, 10)
        c |> decrement()
        c |> decrement()
        print("100 - 20 = {c |> get()}\n")  // 80
    }
```

Building and running

```
cmake --build build --config Release --target integration_cpp_08
bin\Release\integration_cpp_08.exe
```

Expected output:

```
=== Counter ===
initial: 0
after 3 increments: 3
after decrement: 2
after add(10): 12
is_positive: true
is_zero: false
after set_step(5) + increment: 17
after reset: 0
is_zero: true

=== Counter with pipe syntax ===
100 - 3*10 = 70

=== StringBuffer ===
empty: true
content: Hello, World!
length: 13
after clear, empty: true
content: daslang
```

See also:

Full source: `08_methods.cpp`, `08_methods.das`

Previous tutorial: tutorial_integration_cpp_callbacks

Next tutorial: tutorial_integration_cpp_operators_and_properties

5.3.9 C++ Integration: Operators and Properties

This tutorial shows how to bind C++ operators and property accessors to daslang, making custom types feel native. Topics covered:

- `addEquNeq<T>` — binding == and != operators
- Custom arithmetic operators via `addExtern` with operator names
- `addProperty<DAS_BIND_MANAGED_PROP(method)>` — property accessors
- `addPropertyExtConst` — const/non-const property overloads
- `addCtorAndUsing` — exposing C++ constructors to daslang
- Overriding `isPod()` / `hasNonTrivialCtor()` for non-POD types
- Properties vs fields in `ManagedStructureAnnotation`

Prerequisites

- Tutorial 08 completed (tutorial_integration_cpp_methods).
- Familiarity with `DAS_CALL_MEMBER` and `addExtern`.

Operators via `addExtern`

daslang resolves operators by name. To bind a C++ operator, register a function with the operator symbol as its daslang name:

```
Vec3 vec3_add(const Vec3 & a, const Vec3 & b) {
    return { a.x + b.x, a.y + b.y, a.z + b.z };
}

Vec3 vec3_neg(const Vec3 & a) {
    return { -a.x, -a.y, -a.z };
}

// Binary "+"
addExtern<DAS_BIND_FUN(vec3_add),
          SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "+",
    SideEffects::none, "vec3_add")
        ->args({"a", "b"});

// Unary "-"
addExtern<DAS_BIND_FUN(vec3_neg),
          SimNode_ExtFuncCallAndCopyOrMove>(
    *this, lib, "-",
    SideEffects::none, "vec3_neg")
        ->args({"a"});
```

Available operator names: +, -, *, /, %, <<, >>, <, >, <=, >=, &, |, ^.

Note: Functions returning handled types by value require `SimNode_ExtFuncCallAndCopyOrMove` as the second template argument.

`addEquNeq<T>`

The helper `addEquNeq<T>` binds both == and != operators in one call. It requires `operator==` and `operator!=` on the C++ type:

```
struct Vec3 {
    bool operator==(const Vec3 & o) const { ... }
    bool operator!=(const Vec3 & o) const { ... }
};

// In module constructor:
addEquNeq<Vec3>(*this, lib);
```

Properties — addProperty

Properties look like fields but call C++ methods under the hood. Define getter methods on the C++ type, then register them in the annotation:

```
struct Vec3 {
    float x, y, z;
    float length() const { return sqrtf(x*x + y*y + z*z); }
    float lengthSq() const { return x*x + y*y + z*z; }
    bool isZero() const { return x == 0 && y == 0 && z == 0; }
};
```

```
struct Vec3Annotation : ManagedStructureAnnotation<Vec3, false> {
    Vec3Annotation(ModuleLibrary & ml)
        : ManagedStructureAnnotation("Vec3", ml)
    {
        // Regular fields
        addField<DAS_BIND_MANAGED_FIELD(x)>("x", "x");
        addField<DAS_BIND_MANAGED_FIELD(y)>("y", "y");
        addField<DAS_BIND_MANAGED_FIELD(z)>("z", "z");

        // Properties - method calls disguised as field access
        addProperty<DAS_BIND_MANAGED_PROP(length)>("length", "length");
        addProperty<DAS_BIND_MANAGED_PROP(lengthSq)>("lengthSq", "lengthSq");
        addProperty<DAS_BIND_MANAGED_PROP(isZero)>("isZero", "isZero");
    }
};
```

In daslang, properties are accessed with dot syntax just like fields:

```
let v = make_vec3(3.0, 4.0, 0.0)
print("{v.length}\n")     // 5.0 - calls Vec3::length()
print("{v.isZero}\n")     // false - calls Vec3::isZero()
print("{v.x}\n")          // 3.0 - direct field access
```

Const/non-const properties — addPropertyExtConst

When a C++ type has both const and non-const overloads of a method, use `addPropertyExtConst` to bind them as a single property. daslang will call the appropriate overload depending on whether the object is mutable (`var`) or immutable (`let`):

```
struct Vec3 {
    // Non-const - called when the object is mutable
    bool editable() { return true; }
    // Const - called when the object is immutable
    bool editable() const { return false; }
};
```

Register with explicit function-pointer types for both overloads:

```
// Template params: <NonConstSig, &Method, ConstSig, &Method>
addPropertyExtConst<
```

(continues on next page)

(continued from previous page)

```
    bool (Vec3::*)(),       &Vec3::editable,    // non-const
    bool (Vec3::*)() const, &Vec3::editable     // const
>("editable", "editable");
```

In daslang, the property value depends on the variable's mutability:

```
let immutable_v = make_vec3(1.0, 2.0, 3.0)
print("{immutable_v.editable}\n")     // false - const overload

var mutable_v = make_vec3(1.0, 2.0, 3.0)
print("{mutable_v.editable}\n")       // true - non-const overload
```

POD vs non-POD handled types

Whether a handled type requires `unsafe` for local variables depends on the annotation's `isLocal()` method, which defaults to:

```
virtual bool isLocal() const {
    return isPod() && !hasNonTrivialCtor()
        && !hasNonTrivialDtor() && !hasNonTrivialCopy();
}
```

- **POD types** (like `Vec3` — no constructors, no virtual functions): `isLocal()` returns `true` automatically. `let` and `var` work without `unsafe`.
- **Non-POD types** (like a struct with a constructor): `isLocal()` returns `false`, and any local variable — even `let` — gives:

  ```
  error[30108]: local variable of type ... requires unsafe
  ```

Consider a `Color` type with a non-trivial constructor:

```
struct Color {
    float r, g, b, a;
    Color() : r(0), g(0), b(0), a(1) {}  // makes it non-POD

    float brightness() const { return 0.299f*r + 0.587f*g + 0.114f*b; }
};
```

Without annotation overrides, local Color variables need workarounds:

```
// ERROR: let c = Color()  → error[30108] without unsafe!

// Way 1: `using` pattern - safe, constructs on stack via block
using() $(var c : Color#) {
    c.r = 1.0
    c.g = 0.5
    print("{c.brightness}\n")   // 0.5925
}

// Way 2: `unsafe` block
unsafe {
```

(continues on next page)

(continued from previous page)

```
    let c = make_color(1.0, 0.5, 0.0, 1.0)
    print("{c.brightness}\n")   // 0.5925
}
```

addCtorAndUsing — exposing constructors

addCtorAndUsing<T> registers two things:

1. A **constructor function** named after the type — calls placement new in daslang.
2. A **``using`` function** for block-based construction — `using() $(var c : Type#) { ... }`.

```
// In the module constructor:
addCtorAndUsing<Color>(*this, lib, "Color", "Color");
```

The `using` pattern is the safe way to work with non-POD types — the variable lives inside a block, so no `unsafe` is required. The # suffix in the block parameter type (`Color#`) marks it as a temporary value.

Overriding isPod() — SafeColor

If you know a C++ type's non-trivial constructor is harmless (e.g., it just zero-initializes fields), you can override the annotation to tell daslang it is safe for local variables. This avoids the need for `unsafe` or `using`.

We define `SafeColor` as a separate struct with the same layout:

```
struct SafeColor {
    float r, g, b, a;
    SafeColor() : r(0), g(0), b(0), a(1) {}
    float brightness() const { ... }
};

struct SafeColorAnnotation
    : ManagedStructureAnnotation<SafeColor, false>
{
    // ... fields, properties ...

    // Override: treat as POD-safe despite non-trivial ctor
    virtual bool isPod() const override { return true; }
    virtual bool isRawPod() const override { return false; }
    virtual bool hasNonTrivialCtor() const override { return false; }
};
```

Now scripts can use SafeColor without `unsafe`:

```
let sc = SafeColor()          // works - annotation says it's safe
var sc2 = SafeColor()         // also works
sc2.r = 1.0
print("{sc2.brightness}\n")   // 0.299
```

Using operators in daslang

All operators work naturally:

```
options gen2
require tutorial_09_cpp

[export]
def test() {
    let a = make_vec3(1.0, 2.0, 3.0)
    let b = make_vec3(4.0, 5.0, 6.0)

    let c = a + b          // calls vec3_add
    let d = a * 2.0        // calls vec3_mul_scalar
    let e = -a             // calls vec3_neg
    print("a == a: {a == a}\n")   // true
    print("a != b: {a != b}\n")   // true
    print("a.length = {a.length}\n")  // property access
```

Building and running

```
cmake --build build --config Release --target integration_cpp_09
bin\Release\integration_cpp_09.exe
```

Expected output:

```
=== Vec3 basics ===
a = (1, 2, 3)
b = (4, 5, 6)

=== Properties ===
a.length   = 3.7416575
a.lengthSq = 14
a.isZero   = false
zero.isZero = true

=== Const/non-const property ===
Vec3 (POD):
  let  v.editable = false
  var  v.editable = true
Color (non-POD, using pattern):
  using: brightness = 0.5925
Color (non-POD, unsafe):
  unsafe let: brightness = 0.5925
SafeColor (overridden annotation, no unsafe needed):
  let sc = SafeColor() : brightness = 0
  var sc2.brightness    = 0.5925

=== Operators ===
a + b = (5, 7, 9)
b - a = (3, 3, 3)
a * 2 = (2, 4, 6)
```

(continues on next page)

(continued from previous page)

```
-a    = (-1, -2, -3)

=== Equality ===
a == a2: true
a != b:  true
a == b:  false

=== Utility functions ===
dot(a, b) = 32
cross(a, b) = (-3, 6, -3)
normalize(a) = (0.26726124, 0.5345225, 0.8017837)
normalize(a).length = 0.99999994
```

See also:

Full source: `09_operators_and_properties.cpp`, `09_operators_and_properties.das`

Previous tutorial: tutorial_integration_cpp_methods Next tutorial: tutorial_integration_cpp_custom_modules

5.3.10 C++ Integration: Custom Modules

This tutorial shows how to organize C++ bindings into multiple modules with dependencies between them. Topics covered:

- Splitting types and functions across separate modules
- `Module::require()` — looking up modules by name
- `addBuiltinDependency` — declaring module dependencies
- `addConstant` — exporting compile-time constants
- `initDependencies()` — deferred initialization for robust ordering
- `NEED_MODULE` ordering in the host program

Prerequisites

- Tutorial 09 completed (tutorial_integration_cpp_operators_and_properties).
- Familiarity with `ManagedStructureAnnotation` and `addExtern`.

Why multiple modules?

So far, every tutorial has used a single module. In real projects, types and functions often belong to separate logical groups. Splitting them into modules gives scripts fine-grained `require` control and keeps each module focused.

In this tutorial we create two modules:

- **math_types** — defines the `Vec2` type and mathematical constants
- **math_utils** — utility functions that *depend on* `Vec2`

Module A: types and constants

The first module registers a type and several constants:

```
class Module_MathTypes : public Module {
public:
    Module_MathTypes() : Module("math_types") {
        ModuleLibrary lib(this);
        lib.addBuiltInModule();

        addAnnotation(make_smart<Vec2Annotation>(lib));

        addExtern<DAS_BIND_FUN(make_vec2),
                  SimNode_ExtFuncCallAndCopyOrMove>(
            *this, lib, "make_vec2",
            SideEffects::none, "make_vec2")
                ->args({"x", "y"});

        // Constants - appear as `let` in daslang
        addConstant(*this, "PI",      (float)M_PI);
        addConstant(*this, "TWO_PI",  (float)(2.0 * M_PI));
        addConstant(*this, "HALF_PI", (float)(M_PI / 2.0));
        addConstant(*this, "ORIGIN",  "origin_marker");  // string
    }
};
REGISTER_MODULE(Module_MathTypes);
```

`addConstant` creates a module-level `let` constant. It supports numeric types (`int`, `float`, `double`, `uint`, etc.) and strings.

Module B: depends on Module A — `initDependencies()`

The second module uses `Vec2` from `math_types` in its function signatures. Instead of registering everything in the constructor, it uses the **``initDependencies()`` pattern** — the production standard for modules with cross-module dependencies.

Why not the constructor?

Modules are constructed during static initialization (triggered by `REGISTER_MODULE`). At that point, the dependency module may not exist yet — its constructor may not have run. `Module::require()` could return `nullptr`, and the dependency's types would not be registered.

`initDependencies()` is called later by the engine, after all modules are constructed. This guarantees that `Module::require()` can find every loaded module.

The pattern

```
class Module_MathUtils : public Module {
    bool initialized = false;  // guard against double-init
public:
    Module_MathUtils() : Module("math_utils") {
        // Empty - all work deferred to initDependencies().
    }

    bool initDependencies() override {
        if (initialized) return true;          // already done

        // 1. Look up the dependency by name
        auto mod = Module::require("math_types");
        if (!mod) return false;                // not loaded

        // 2. Ensure it is fully initialized
        if (!mod->initDependencies()) return false;

        // 3. Mark ourselves as initialized (before registration
        //    to prevent re-entry from circular dependencies)
        initialized = true;

        // 4. Set up library with dependency
        ModuleLibrary lib(this);
        lib.addBuiltInModule();
        addBuiltinDependency(lib, mod);

        // 5. Register functions - Vec2 is now known
        addExtern<DAS_BIND_FUN(vec2_length)>(
            *this, lib, "length", ...);
        addExtern<DAS_BIND_FUN(vec2_lerp), ...>(
            *this, lib, "lerp", ...);
        // ...

        return true;
    }
};
REGISTER_MODULE(Module_MathUtils);
```

Key elements:

`initialized` guard	Prevents double-initialization
`Module::require("name")`	Looks up module by registered name
`mod->initDependencies()`	Ensures dependency is fully registered
`addBuiltinDependency`	Makes dependency's types visible + records the relationship for the compiler
`return true/false`	Signals success or failure to the engine

Real-world examples

This pattern appears throughout the daslang ecosystem:

dasAudio (requires `rtti`):

```
class Module_Audio : public Module {
    bool initialized = false;
public:
    Module_Audio() : Module("audio") {}
    bool initDependencies() override {
        if (initialized) return true;
        if (!Module::require("rtti")) return false;
        initialized = true;
        ModuleLibrary lib(this);
        lib.addBuiltInModule();
        addBuiltinDependency(lib, Module::require("rtti"));
        // ... register types, functions ...
        return true;
    }
};
```

dasIMGUI_NODE_EDITOR (requires `imgui`):

```
bool Module_dasIMGUI_NODE_EDITOR::initDependencies() {
    if (initialized) return true;
    auto mod_imgui = Module::require("imgui");
    if (!mod_imgui) return false;
    if (!mod_imgui->initDependencies()) return false;
    initialized = true;
    lib.addModule(this);
    lib.addBuiltInModule();
    lib.addModule(mod_imgui);
    // ... register types, functions ...
    return true;
}
```

The `mod->initDependencies()` call is especially important — it chains initialization so that modules initialize in the correct order regardless of how `NEED_MODULE` lines are arranged.

Constructor vs `initDependencies()` — when to use which

Constructor	Leaf modules with no custom-module deps (like `math_types` above)
``initDependencies()``	Any module that depends on another custom module — use this by default

Host program — `NEED_MODULE`

In `main()`, both modules must be listed. With `initDependencies()`, explicit ordering is no longer critical — the chained `mod->initDependencies()` calls handle it automatically:

```
int main(int, char * []) {
    NEED_ALL_DEFAULT_MODULES;
    NEED_MODULE(Module_MathTypes);
    NEED_MODULE(Module_MathUtils);
    Module::Initialize();
    // ... compile & run ...
    Module::Shutdown();
    return 0;
}
```

`NEED_MODULE` forces the linker to pull in the module's translation unit. `Module::require("name")` finds a module by its registered name string, returning `nullptr` if not found.

Using both modules from daslang

Scripts `require` each module independently:

```
options gen2
require math_types
require math_utils

[export]
def test() {
    // Constants from math_types
    print("PI = {PI}\n")

    // Type from math_types
    let a = make_vec2(3.0, 4.0)

    // Functions from math_utils (operate on Vec2)
    print("length(a) = {length(a)}\n")

    let mid = lerp(a, make_vec2(1.0, 0.0), 0.5)
    print("mid = ({mid.x}, {mid.y})\n")
```

Building and running

```
cmake --build build --config Release --target integration_cpp_10
bin\Release\integration_cpp_10.exe
```

Expected output:

```
=== Constants ===
PI      = 3.1415927
TWO_PI  = 6.2831855
HALF_PI = 1.5707964
ORIGIN  = origin_marker

=== Vec2 ===
a = (3, 4)
b = (1, 0)

=== Utility functions ===
length(a)     = 5
dot(a, b)     = 3
normalize(a)  = (0.6, 0.8)

=== Operators ===
a + b = (4, 4)
a * 2 = (6, 8)

=== Lerp ===
lerp(a, b, 0.5) = (2, 2)
lerp(a, b, 0.0) = (3, 4)
lerp(a, b, 1.0) = (1, 0)
```

See also:

Full source: `10_custom_modules.cpp`, `10_custom_modules.das`

Previous tutorial: tutorial_integration_cpp_operators_and_properties

Next tutorial: tutorial_integration_cpp_context_variables

5.3.11 C++ Integration: Context Variables

This tutorial shows how to read and write daslang global variables from C++ host code. Topics covered:

- `ctx.findVariable()` — look up globals by name
- `ctx.getVariable()` — get a raw pointer to variable data
- Reading and writing scalar, string, and struct globals
- `ctx.getTotalVariables()` and `ctx.getVariableInfo()` — enumeration

Prerequisites

- Tutorial 10 completed (tutorial_integration_cpp_custom_modules).
- Familiarity with `ManagedStructureAnnotation`.

Global variable layout

After `program->simulate(ctx, tout)`, global variables live in a contiguous memory buffer inside the `Context`. The `Context` provides methods to find variables by name and access their raw data:

```
// Find by name → index (returns -1 if not found)
int idx = ctx.findVariable("score");

// Index → raw pointer to data
void * ptr = ctx.getVariable(idx);

// Cast and use
int32_t value = *(int32_t *)ptr;    // read
*(int32_t *)ptr = 9999;             // write - visible to daslang!
```

Reading scalar globals

```
int idx_score = ctx.findVariable("score");
if (idx_score >= 0) {
    int32_t * pScore = (int32_t *)ctx.getVariable(idx_score);
    printf("score = %d\n", *pScore);
}

int idx_name = ctx.findVariable("player_name");
if (idx_name >= 0) {
    // String globals are stored as char*
    char ** pName = (char **)ctx.getVariable(idx_name);
    printf("player_name = %s\n", *pName);
}
```

Reading struct globals

If the script has a global of a handled type, the pointer points directly to the C++ struct:

```
int idx = ctx.findVariable("config");
if (idx >= 0) {
    GameConfig * p = (GameConfig *)ctx.getVariable(idx);
    printf("gravity = %.1f\n", p->gravity);
}
```

Writing globals from C++

Any changes made through the raw pointer are immediately visible to daslang code:

```
*(int32_t *)ctx.getVariable(idx_score) = 9999;

// Call a daslang function - it will see score == 9999
auto fn = ctx.findFunction("print_globals");
ctx.evalWithCatch(fn, nullptr);
```

Enumerating all variables

`getTotalVariables()` returns the count, `getVariableInfo()` returns a `VarInfo *` with name and size:

```
int total = ctx.getTotalVariables();
for (int i = 0; i < total; i++) {
    auto * info = ctx.getVariableInfo(i);
    if (info) {
        printf("[%d] %s (size=%d)\n", i, info->name, info->size);
    }
}
```

Initialization

`simulate()` automatically calls `runInitScript()` internally, executing global initializers and `[init]` functions. No separate initialization step is needed — globals have their initial values immediately after simulation.

The daslang side

```
options gen2
require tutorial_11_cpp

var score : int = 42
var player_name : string = "Hero"
var alive : bool = true
var config = GameConfig(gravity = 9.8, speed = 5.0, max_enemies = 10)

[export]
def print_globals() {
    print("score = {score}\n")
    // C++ changes are visible here
```

Building and running

```
cmake --build build --config Release --target integration_cpp_11
bin\Release\integration_cpp_11.exe
```

Expected output:

```
=== Reading globals from C++ ===
score = 42
player_name = Hero
alive = true
config.gravity     = 9.8
config.speed       = 5.0
config.max_enemies = 10

=== Writing globals from C++ ===
C++ set score = 9999
C++ set config.gravity = 20.0, max_enemies = 100

=== Calling daslang to verify ===
  score        = 9999
  player_name  = Hero
  alive        = true
  config.gravity     = 20
  config.speed       = 5
  config.max_enemies = 100

=== All global variables ===
  [0] score (size=4)
  [1] player_name (size=8)
  [2] alive (size=1)
  [3] config (size=12)
```

See also:

Full source: `11_context_variables.cpp`, `11_context_variables.das`

Previous tutorial: tutorial_integration_cpp_custom_modules

Next tutorial: tutorial_integration_cpp_smart_pointers

5.3.12 C++ Integration: Smart Pointers

This tutorial shows how to expose reference-counted C++ types to daslang using `smart_ptr<T>`. Topics covered:

- Inheriting from `das::ptr_ref_count` for reference counting
- `ManagedStructureAnnotation` with `canNew` / `canDelete`
- `var inscope` — automatic cleanup of smart pointers
- Factory functions returning `smart_ptr<T>`
- `smart_ptr_clone` and `smart_ptr_use_count`
- `new T` for heap allocation from scripts

Prerequisites

- Tutorial 11 completed (tutorial_integration_cpp_context_variables).
- Understanding of reference counting concepts.

Making a type reference-counted

A C++ type becomes smart-pointer-compatible by inheriting from `das::ptr_ref_count`, which provides `addRef()`, `delRef()`, and `use_count()`:

```
#include "daScript/daScript.h"

class Entity : public das::ptr_ref_count {
public:
    das::string name;
    float x, y;
    int32_t health;

    Entity() : name("unnamed"), x(0), y(0), health(100) {
        printf("  Entity constructed\n");
    }
    ~Entity() {
        printf("  Entity destroyed\n");
    }
    // ... methods ...
};
```

When `delRef()` decrements the count to zero, the object is automatically deleted.

Annotation — `canNew` and `canDelete`

`ManagedStructureAnnotation` takes two boolean template parameters that control what scripts can do:

```
struct EntityAnnotation
    : ManagedStructureAnnotation<Entity, true, true>
    //                                   ^^^^  ^^^^
    //                           canNew ---+      |
    //                           canDelete -------+
{
    EntityAnnotation(ModuleLibrary & ml)
        : ManagedStructureAnnotation("Entity", ml)
    {
        addField<DAS_BIND_MANAGED_FIELD(name)>("name", "name");
        addField<DAS_BIND_MANAGED_FIELD(x)>("x", "x");
        addField<DAS_BIND_MANAGED_FIELD(y)>("y", "y");
        addField<DAS_BIND_MANAGED_FIELD(health)>("health", "health");
        addProperty<DAS_BIND_MANAGED_PROP(is_alive)>(
            "is_alive", "is_alive");
    }
};
```

`isSmart()` is auto-detected — `ManagedStructureAnnotation` checks `is_base_of<ptr_ref_count, Entity>` at compile time.

canNew	new Entity allocates + addRef()
canDelete	delete ptr calls delRef()

Factory returning smart_ptr<T>

The typical pattern is a factory function that returns smart_ptr:

```
smart_ptr<Entity> make_entity(const char * name, float x, float y) {
    auto e = make_smart<Entity>();
    e->name = name;
    e->x = x;
    e->y = y;
    return e;
}

// Registration:
addExtern<DAS_BIND_FUN(make_entity)>(*this, lib, "make_entity",
    SideEffects::modifyExternal, "make_entity")
        ->args({"name", "x", "y"});
```

Using smart pointers in daslang

Smart pointer variables must be declared with var inscope, which ensures delRef() is called when the variable goes out of scope:

```
options gen2
require tutorial_12_cpp

[export]
def test() {
    // Factory returns smart_ptr<Entity> - use <- to move
    var inscope hero <- make_entity("Hero", 0.0, 0.0)
    print("{hero.name} hp={hero.health}\n")

    // Dereference with * when passing to functions taking Entity &
    move_entity(*hero, 3.0, 4.0)

    // Clone increases reference count
    var inscope hero2 : smart_ptr<Entity>
    smart_ptr_clone(hero2, hero)
    print("use_count = {int(smart_ptr_use_count(hero))}\n")  // 2

    // `new Entity` - heap allocation (requires unsafe)
    unsafe {
        var inscope fresh <- new Entity
        fresh.health = 50
    }
    // fresh destroyed here (delRef → ref_count==0 → delete)
```

Key points:

- var inscope — required for smart_ptr variables

- <- — move semantics (not =)
- *ptr — dereference to get Entity & for function calls
- smart_ptr_clone(dest, src) — clone (addRef)
- smart_ptr_use_count(ptr) — returns uint (cast to int for decimal printing)
- new Entity — allocates + addRef (needs unsafe)

Building and running

```
cmake --build build --config Release --target integration_cpp_12
bin\Release\integration_cpp_12.exe
```

Expected output:

```
=== Factory-created entities ===
  [C++] Entity('unnamed') constructed
  [C++] Entity('unnamed') constructed
hero:  Hero at (0, 0) hp=100
enemy: Goblin at (10, 5) hp=100
use_count(hero)  = 1
use_count(enemy) = 1

=== Moving and combat ===
hero moved to (3, 4)
distance = 7.071068
enemy hp after 30 damage = 70
enemy.is_alive = true
enemy hp after 100 more  = 0
enemy.is_alive = false

=== Cloning smart_ptr ===
After clone:
  use_count(hero)  = 2
  use_count(hero2) = 2
  hero2.name = Hero

=== new Entity() ===
  [C++] Entity('unnamed') constructed
fresh.name   = Newbie
fresh.health = 50
use_count    = 1
  [C++] Entity('Newbie') destroyed

=== End of test (hero, enemy, hero2 destroyed here) ===
  [C++] Entity('Goblin') destroyed
  [C++] Entity('Hero') destroyed
```

See also:

Full source: 12_smart_pointers.cpp, 12_smart_pointers.das

Previous tutorial: tutorial_integration_cpp_context_variables

Next tutorial: tutorial_integration_cpp_aot

5.3.13 C++ Integration: AOT Compilation

This tutorial explains daslang's ahead-of-time (AOT) compilation system, which transpiles daslang functions into C++ source code for near-native performance. Topics covered:

- The AOT workflow — generate, compile, link
- `CodeOfPolicies::aot` — enabling AOT at runtime
- `SimFunction::aot` — checking if a function is AOT-linked
- Self-registration via `AotListBase`

Prerequisites

- Tutorial 12 completed (tutorial_integration_cpp_smart_pointers).
- Understanding of the compile → simulate → eval workflow.

What AOT does

AOT in daslang does **not** JIT-compile to machine code at runtime. Instead, it follows a **two-stage build** pattern:

1. **Generate** — the `daslang` tool transpiles daslang functions into C++ source code
2. **Compile** — the generated `.cpp` file is compiled by your C++ compiler and linked into the host executable
3. **Link** — at runtime, `simulate()` replaces interpreted simulation nodes with direct calls to the pre-compiled C++ functions

This gives near-C++ performance while keeping the convenience of scripting during development.

The AOT workflow

Stage 1 — Generate C++ from daslang

The project's `CMakeLists.txt` uses the `DAS_AOT` macro, which runs `daslang.exe` with the AOT tool script at build time:

```
daslang.exe utils/aot/main.das -- -aot script.das output.cpp
```

This produces a `.cpp` file containing C++ implementations of all daslang functions, plus a self-registration block:

```
// Generated AOT code (simplified)
namespace das {
    namespace _anon_XXX {
        inline bool test(Context * __context__) {
            // ... translated daslang code ...
        }

        static void registerAotFunctions(AotLibrary & aotLib) {
            aotLib[0x92da443ef141abbb] = +[](Context & ctx) -> SimNode* {
                return ctx.code->makeNode<SimNode_Aot<...>>();
            };
        }
    }
```

(continues on next page)

(continued from previous page)

```
    AotListBase impl(registerAotFunctions);  // self-registers
}
```

The `AotListBase` constructor runs at program startup, inserting the registration function into a global linked list.

Stage 2 — Compile into the host

The `CMakeLists.txt` for this tutorial uses the `DAS_AOT` macro to automate both generation and compilation:

```
# 1. Create a custom target and accumulator variable
SET(INTEGRATION_13_AOT_GENERATED_SRC)
add_custom_target(integration_cpp_13_dasAotStub)

# 2. Generate AOT C++ from the .das file using daslang as the tool
DAS_AOT("tutorials/integration/cpp/13_aot.das"
        INTEGRATION_13_AOT_GENERATED_SRC
        integration_cpp_13_dasAotStub daslang)

# 3. Build the executable with both hand-written AND generated sources
add_executable(integration_cpp_13
    13_aot.cpp
    ${INTEGRATION_13_AOT_GENERATED_SRC}
)
TARGET_LINK_LIBRARIES(integration_cpp_13 libDaScript Threads::Threads)
ADD_DEPENDENCIES(integration_cpp_13 libDaScript
                 integration_cpp_13_dasAotStub)
```

The `DAS_AOT` macro:

- Runs `daslang.exe utils/aot/main.das -- -aot 13_aot.das output.cpp` as a custom build command
- Puts the generated file in an `_aot_generated/` subdirectory
- Creates a dependency on `daslang` so it's built first

The generated `.cpp` is compiled alongside `13_aot.cpp` — no manual steps required.

Stage 3 — Enable AOT in the host

```
CodeOfPolicies policies;
policies.aot = true;              // enable AOT linking
policies.fail_on_no_aot = true;   // error if any function lacks AOT

auto program = compileDaScript(scriptPath, fAccess, tout,
                               libGroup, policies);

Context ctx(program->getContextStackSize());
program->simulate(ctx, tout);  // AOT functions linked here

auto fn = ctx.findFunction("test");
```

(continues on next page)

(continued from previous page)

```
// Check if AOT-linked:
if (fn->aot) { /* running native C++ */ }
```

During `simulate()`, the engine computes a semantic hash for each function, looks it up in the global AOT library, and replaces the simulation node if a match is found.

Checking AOT status

`SimFunction` (returned by `ctx.findFunction()`) has an `aot` flag:

```
auto fn = ctx.findFunction("test");
printf("AOT: %s\n", fn->aot ? "yes" : "no");
```

Without the generated `.cpp` linked in, `aot` is `false` and the function runs through the interpreter. With it linked in, `aot` is `true`.

Key policies

`policies.aot`	Enable AOT linking during simulate
`policies.fail_on_no_aot`	Error if a function lacks AOT (default `true`)
`policies.aot_module`	This is a module AOT (not entry point)

Building and running

```
cmake --build build --config Release --target integration_cpp_13
bin\Release\integration_cpp_13.exe
```

Expected output:

```
=== Interpreter mode ===
  test() is AOT: no
=== AOT Tutorial ===
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34

sin(pi/4) = 0.70710677
cos(pi/4) = 0.70710677
sqrt(2)   = 1.4142135

=== AOT mode (policies.aot = true) ===
  test() is AOT: yes
```

(continues on next page)

(continued from previous page)

```
=== AOT Tutorial ===
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34

sin(pi/4) = 0.70710677
cos(pi/4) = 0.70710677
sqrt(2)   = 1.4142135

=== AOT workflow summary ===
1. daslang.exe -aot script.das script.das.cpp
   -> generates C++ source from daslang functions
2. Compile the .cpp into your executable
   -> functions self-register via static AotListBase
3. Set CodeOfPolicies::aot = true before compileDaScript
   -> simulate() links AOT functions automatically
4. Functions run as native C++ - no interpreter overhead
```

Without `policies.aot`, the function runs through the interpreter (`test() is AOT: no`). With it set, the pre-compiled C++ code is linked in (`test() is AOT: yes`) for near-native performance.

See also:

Full source: `13_aot.cpp`, `13_aot.das`

Previous tutorial: tutorial_integration_cpp_smart_pointers

Next tutorial: tutorial_integration_cpp_serialization

5.3.14 C++ Integration: Serialization

This tutorial shows how to serialize a compiled daslang program to a binary blob and deserialize it, skipping recompilation on subsequent runs. Topics covered:

- `AstSerializer` — the serializer/deserializer
- `SerializationStorageVector` — in-memory serialization buffer
- `Program::serialize()` — serializing a compiled program
- The compile → serialize → deserialize → simulate workflow

Prerequisites

- Tutorial 13 completed (tutorial_integration_cpp_aot).

Why serialize?

Compiling a daslang program involves parsing, type inference, and optimization — this can take noticeable time for large scripts. Serialization saves the compiled program to a binary blob so that future runs can skip compilation entirely:

1. **First run** — compile from source, serialize to file/buffer
2. **Subsequent runs** — deserialize from blob, simulate, run

This is useful for:

- Faster startup in production applications
- Distributing pre-compiled scripts
- Caching compiled programs in editors and build pipelines

Required header

```
#include "daScript/ast/ast_serializer.h"
```

The serialization workflow

Stage 1 — Compile from source

```
auto program = compileDaScript(scriptPath, fAccess, tout, libGroup);
// Must simulate once before serializing
Context ctx(program->getContextStackSize());
program->simulate(ctx, tout);
```

Stage 2 — Serialize to memory

```
auto writeTo = make_unique<SerializationStorageVector>();
{
    AstSerializer ser(writeTo.get(), true);  // writing = true
    program->serialize(ser);
    ser.moduleLibrary = nullptr;
}
size_t blobSize = writeTo->buffer.size();
program.reset();  // release original program
```

`SerializationStorageVector` wraps a `vector<uint8_t>` buffer. After serialization, the buffer contains the entire compiled program.

Stage 3 — Deserialize

```
auto readFrom = make_unique<SerializationStorageVector>();
readFrom->buffer = das::move(writeTo->buffer);
{
    AstSerializer deser(readFrom.get(), false);  // writing = false
    program = make_smart<Program>();
    program->serialize(deser);
    deser.moduleLibrary = nullptr;
}
```

Note that `Program::serialize()` is used for both directions — the `AstSerializer` constructor's `isWriting` flag determines the mode.

Stage 4 — Simulate and run

```
Context ctx(program->getContextStackSize());
program->simulate(ctx, tout);
auto fn = ctx.findFunction("test");
ctx.evalWithCatch(fn, nullptr);
```

The deserialized program works identically to a freshly compiled one.

Saving to and loading from disk

To persist across application restarts, write the buffer to a file:

```
// Save
FILE * f = fopen("script.cache", "wb");
fwrite(writeTo->buffer.data(), 1, writeTo->buffer.size(), f);
fclose(f);

// Load
auto readFrom = make_unique<SerializationStorageVector>();
FILE * f2 = fopen("script.cache", "rb");
fseek(f2, 0, SEEK_END);
size_t sz = ftell(f2);
fseek(f2, 0, SEEK_SET);
readFrom->buffer.resize(sz);
fread(readFrom->buffer.data(), 1, sz, f2);
fclose(f2);
```

Building and running

```
cmake --build build --config Release --target integration_cpp_14
bin\Release\integration_cpp_14.exe
```

Expected output:

```
=== Stage 1: Compile from source ===
  Compiled successfully.
  Simulated successfully.

=== Stage 2: Serialize ===
  Serialized size: 5022 bytes
  Original program released.

=== Stage 3: Deserialize ===
  Deserialized successfully.

=== Stage 4: Simulate and run ===
=== Serialization Tutorial ===
Hello, World!
sum_to(10)  = 55
sum_to(100) = 5050
```

See also:

Full source: `14_serialization.cpp`, `14_serialization.das`

Previous tutorial: tutorial_integration_cpp_aot

Next tutorial: tutorial_integration_cpp_custom_annotations

5.3.15 C++ Integration: Custom Annotations

This tutorial shows how to create custom annotations in C++ that modify daslang compilation behavior. Topics covered:

- `FunctionAnnotation` — hooks into function compilation
- `StructureAnnotation` — hooks into struct compilation
- `apply()` / `finalize()` — the function annotation lifecycle
- `touch()` / `look()` — the structure annotation lifecycle
- Adding fields to structs at compile time

Prerequisites

- Tutorial 14 completed (tutorial_integration_cpp_serialization).

What are annotations?

Annotations are compile-time hooks defined in C++ and used from daslang with the `[annotation_name]` syntax. They let the host application validate, modify, or transform script code during compilation.

Built-in examples: `[export]`, `[private]`, `[deprecated]`. This tutorial shows how to create your own.

Annotation class hierarchy

```
Annotation
├── FunctionAnnotation      - [name] on functions
│   └── MarkFunctionAnnotation  - convenience base
├── StructureAnnotation     - [name] on structs
└── TypeAnnotation          - describes C++ types
    └── ManagedStructureAnnotation<T>
```

`TypeAnnotation` / `ManagedStructureAnnotation` describe **how a C++ type is exposed** (fields, size, simulation nodes). `StructureAnnotation` is a **compile-time hook** on daslang `struct` declarations — they are unrelated concepts.

FunctionAnnotation — `[log_calls]`

A `FunctionAnnotation` is called during compilation whenever the annotated function is parsed and type-checked:

```
struct LogCallsAnnotation : FunctionAnnotation {
    LogCallsAnnotation() : FunctionAnnotation("log_calls") {}

    // Called during parsing - can modify function flags
    bool apply(const FunctionPtr & func, ModuleGroup &,
               const AnnotationArgumentList &, string &) override {
        printf("[log_calls] apply: %s\n", func->name.c_str());
        return true;  // true = success, false = error
    }

    // Not supported on blocks
    bool apply(ExprBlock *, ModuleGroup &,
               const AnnotationArgumentList &,
               string & err) override {
        err = "not supported for blocks";
        return false;
    }

    // Called after type inference
    bool finalize(const FunctionPtr & func, ModuleGroup &,
                  const AnnotationArgumentList &,
                  const AnnotationArgumentList &,
                  string &) override {
        printf("[log_calls] finalize: %s (args: %d)\n",
               func->name.c_str(), (int)func->arguments.size());
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    bool finalize(ExprBlock *, ModuleGroup &,
                  const AnnotationArgumentList &,
                  const AnnotationArgumentList &,
                  string &) override { return true; }
};
```

Key virtual methods:

`apply()`	Parse time — modify flags, validate
`finalize()`	After inference — validate with types
`verifyCall()`	Each call site — check arguments
`transformCall()`	Inference — rewrite call AST
`simulate()`	Simulation — return custom SimNode

StructureAnnotation — `[add_field]`

A `StructureAnnotation` is called during compilation of annotated structs. The key hook is `touch()`, which runs **before** type inference and can modify the struct:

```
struct AddFieldAnnotation : StructureAnnotation {
    AddFieldAnnotation() : StructureAnnotation("add_field") {}

    // Called BEFORE type inference - can modify struct
    bool touch(const StructurePtr & st, ModuleGroup &,
               const AnnotationArgumentList &, string &) override {
        if (!st->findField("id")) {
            st->fields.emplace_back(
                "id",                               // name
                make_smart<TypeDecl>(Type::tInt),  // type
                nullptr,                            // no init
                AnnotationArgumentList(),           // no ann
                false,                              // no move
                LineInfo()                          // loc
            );
            // Must invalidate the lookup cache!
            st->fieldLookup.clear();
        }
        return true;
    }

    // Called AFTER type inference - read-only validation
    bool look(const StructurePtr & st, ModuleGroup &,
              const AnnotationArgumentList &,
              string &) override {
        printf("struct '%s' has %d fields\n",
               st->name.c_str(), (int)st->fields.size());
        return true;
    }
};
```

Key points:

- Push to `st->fields` directly — there is no `addField` method
- **Must call** `st->fieldLookup.clear()` after adding fields
- Guard with `findField()` to avoid duplicates (`touch` can be called multiple times)
- `touch` → before inference (can modify)
- `look` → after inference (read-only)
- `patch` → after inference (can request re-inference)

Registration

Register annotations in the module constructor:

```
addAnnotation(make_smart<LogCallsAnnotation>());
addAnnotation(make_smart<AddFieldAnnotation>());
```

Using from daslang

```
options gen2
require tutorial_15_cpp

[log_calls]
def attack(target : string; damage : int) {
    print("  {target} takes {damage} damage!\n")
}

[add_field]
struct Monster {
    name : string
    hp   : int
}

[export]
def test() {
    attack("Goblin", 25)
    // Monster now has an "id" field added by [add_field]
    var m = Monster(name = "Dragon", hp = 500, id = 42)
    print("{m.name}: id={m.id}\n")
```

Building and running

```
cmake --build build --config Release --target integration_cpp_15
bin\Release\integration_cpp_15.exe
```

Expected output:

```
--- Compilation (annotation hooks fire here) ---
  [log_calls] apply:    attack
  [log_calls] apply:    heal
```

(continues on next page)

(continued from previous page)

```
  [add_field] added 'id : int' to struct 'Monster'
  [add_field] look: struct 'Monster' has 3 fields
  [log_calls] finalize: attack (args: 2)
  [log_calls] finalize: heal (args: 2)

--- Running script ---
=== Custom Annotations Tutorial ===

--- Annotated functions ---
  Goblin takes 25 damage!
  Hero heals for 10 hp

--- Normal function ---
  (this function has no annotation)

--- Annotated struct ---
  Dragon: hp=500, id=42
```

Note how the annotation hooks fire during **compilation**, before the script runs.

See also:

Full source: `15_custom_annotations.cpp`, `15_custom_annotations.das`

Previous tutorial: tutorial_integration_cpp_serialization

Next tutorial: tutorial_integration_cpp_sandbox

5.3.16 C++ Integration: Sandbox

This tutorial shows how to restrict what daslang scripts can do, which is essential when running untrusted code (user mods, plugin systems, online playgrounds). Two complementary approaches are shown:

- **Approach A — C++**: `CodeOfPolicies` + `FileAccess` subclass
- **Approach B — .das_project**: sandbox policy written in daslang

Topics covered:

- `CodeOfPolicies` — compile-time language restrictions
- Memory limits — heap and stack size caps
- Custom `FileAccess` — restrict which modules scripts can require
- `.das_project` — policy file with exported callback functions

Prerequisites

- Tutorial 15 completed (tutorial_integration_cpp_custom_annotations).

Sandboxing approaches

daslang provides two complementary approaches:

Approach A — C++ code (Demos 1–4):

1. `CodeOfPolicies` — language-level restrictions passed to `compileDaScript()`
2. `FileAccess` virtual overrides — fine-grained C++ control over modules, options, annotations

Approach B — .das_project file (Demos 5–6):

3. A `.das_project` file — a regular daslang script that exports callback functions. The host loads it, and the compiler calls the callbacks during compilation of user scripts.

Both can be combined — `CodeOfPolicies` is layered on top of either `FileAccess` subclass or `.das_project` callbacks.

CodeOfPolicies — language restrictions

Pass a `CodeOfPolicies` struct to `compileDaScript()`:

```
CodeOfPolicies policies;
policies.no_unsafe = true;                    // forbid unsafe blocks
policies.no_global_variables = true;          // forbid module-scope var
policies.no_init = true;                      // forbid [init] functions
policies.max_heap_allocated = 1024 * 1024;    // 1 MB heap max
policies.max_string_heap_allocated = 256*1024; // 256 KB strings
policies.stack = 8 * 1024;                    // 8 KB stack

auto program = compileDaScript(script, fAccess, tout,
                               libGroup, policies);
```

Key policy flags:

`no_unsafe`	Forbids all `unsafe` blocks
`no_global_variables`	Forbids module-scope `var`
`no_global_heap`	Forbids heap allocations from globals
`no_init`	Forbids `[init]` functions
`max_heap_allocated`	Caps heap memory (0 = unlimited)
`max_string_heap_allocated`	Caps string heap memory
`stack`	Context stack size in bytes
`threadlock_context`	Adds context mutex for thread safety

When a policy is violated, compilation fails with error `40207`.

Custom FileAccess — module restrictions

Subclass `FsFileAccess` and override virtual methods to control what scripts can access:

```
class SandboxFileAccess : public FsFileAccess {
    das_set<string> allowedModules;
public:
    SandboxFileAccess() {
        allowedModules.insert("$");      // built-in (always needed)
        allowedModules.insert("math");
        allowedModules.insert("strings");
    }

    bool isModuleAllowed(const string & mod,
                         const string &) const override {
        return allowedModules.count(mod) > 0;
    }

    bool canModuleBeUnsafe(const string &,
                           const string &) const override {
        return false;  // no module can use unsafe
    }

    bool isOptionAllowed(const string & opt,
                         const string &) const override {
        return opt != "persistent_heap";
    }
};
```

Available virtual overrides:

`isModuleAllowed()`	Can this module be loaded at all?
`canModuleBeUnsafe()`	Can this module use `unsafe`?
`canBeRequired()`	Can this module be `require`'d?
`isOptionAllowed()`	Can this `options` keyword be used?
`isAnnotationAllowed()`	Can this annotation be used?

All return `true` by default (no restrictions).

.das_project — policy in daslang

Instead of writing C++ code, you can define sandbox policies in a `.das_project` file — a regular `.das` script that exports callback functions. The compiler loads it, simulates it, and calls the exported functions during compilation of user scripts.

Loading a project file from C++

```
string projectPath = getDasRoot()
    + "/path/to/sandbox.das_project";
auto fAccess = make_smart<FsFileAccess>(
    projectPath, make_smart<FsFileAccess>());

CodeOfPolicies policies;
auto program = compileDaScript(scriptPath, fAccess, tout,
                               libGroup, policies);
```

The `FsFileAccess(projectPath, fallbackAccess)` constructor compiles and simulates the project file, then looks up exported callback functions by name.

Project file callbacks

```
options gen2
require strings
require daslib/strings_boost

typedef module_info = tuple<string; string; string> const
var DAS_PAK_ROOT = "./"

// REQUIRED - resolve every `require X` to a file path
[export]
def module_get(req, from : string) : module_info {
    // return (moduleName, fileName, importName)
}

// Whitelist which modules can be loaded
[export]
def module_allowed(mod, filename : string) : bool {
    if (mod == "$" || mod == "math" || mod == "strings") {
        return true
    }
    return false
}

// No module may use unsafe
[export]
def module_allowed_unsafe(mod, filename : string) : bool {
    return false
}

// Only safe options
```

(continues on next page)

(continued from previous page)

```
[export]
def option_allowed(opt, from : string) : bool {
    return opt == "gen2" || opt == "indenting"
}

// Only safe annotations
[export]
def annotation_allowed(ann, from : string) : bool {
    return ann == "export" || ann == "private"
}
```

Available callbacks:

`module_get`	Resolve `require` paths (REQUIRED)
`module_allowed`	Whitelist which modules can load
`module_allowed_unsafe`	Control `unsafe` per module
`option_allowed`	Whitelist `options` directives
`annotation_allowed`	Whitelist annotations
`include_get`	Resolve `include` directives (optional)

`DAS_PAK_ROOT` is set by the runtime to the directory containing the `.das_project` file, useful for resolving relative paths.

C++ vs .das_project — when to use which

- **C++ subclass** — when you need dynamic logic, access to host state, or tight integration with your engine's asset pipeline
- **.das_project** — when you want to ship policy as data alongside scripts, let level designers tweak restrictions, or prototype policies without recompiling the host
- **Both combined** — `CodeOfPolicies` plus a `.das_project`; the C++ policies add a hard floor, the project file refines what's allowed within that floor

Demo walkthrough

The tutorial runs six demos:

Approach A — C++:

1. **No restrictions** — normal compilation succeeds
2. **``no_unsafe`` policy** — a script with `unsafe` blocks is rejected; a safe script compiles fine
3. **Memory limits** — stack and heap sizes are capped
4. **Module restrictions** — `SandboxFileAccess` blocks modules not in the allow-list; `canModuleBeUnsafe() = false` prevents transitive dependencies from using unsafe

Approach B — .das_project:

5. **.das_project sandbox** — the safe script compiles under the project's policy callbacks
6. **.das_project blocks violations** — unsafe scripts and blocked modules are rejected by the project's callbacks

Building and running

```
cmake --build build --config Release --target integration_cpp_16
bin\Release\integration_cpp_16.exe
```

Expected output:

```
=== Demo 1: No restrictions ===
--- Normal mode ---
=== Sandbox Tutorial ===
score = 42
sum(1..10) = 55
Hello from the sandbox!

=== Demo 2: Policies - no_unsafe ===
--- Safe script with no_unsafe ---
=== Sandbox Tutorial ===
...

--- Unsafe script with no_unsafe ---
  Compilation FAILED (expected in sandbox demo):
  error[40207]: unsafe function test
  ...

=== Demo 3: Memory limits ===
  Stack:       8192 bytes
  Max heap:    1048576 bytes
  Max strings: 262144 bytes
--- Memory-limited ---
=== Sandbox Tutorial ===
...

=== Demo 4: Module restrictions ===
--- Allowed script ---
=== Sandbox Tutorial ===
...

--- Blocked module script ---
  Compilation FAILED (expected in sandbox demo):
  ...

=== Demo 5: .das_project sandbox ===
--- Script under .das_project sandbox ---
=== Sandbox Tutorial ===
score = 42
sum(1..10) = 55
Hello from the sandbox!

=== Demo 6: .das_project blocks violations ===
--- Unsafe script under .das_project ---
  Compilation FAILED (expected in sandbox demo):
  error[40207]: unsafe function test
  ...
```

(continues on next page)

(continued from previous page)

```
--- Blocked module under .das_project ---
  Compilation FAILED (expected in sandbox demo):
  ...

=== Available sandbox approaches ===
...
```

See also:

Full source: `16_sandbox.cpp`, `16_sandbox.das`, `16_sandbox.das_project`

Previous tutorial: tutorial_integration_cpp_custom_annotations

Next tutorial: tutorial_integration_cpp_coroutines

5.3.17 C++ Integration: Coroutines

This tutorial shows how to consume a daslang **generator** (coroutine) from C++. A daslang function returns a `generator<int>` via `return <-`. The C++ host receives a `Sequence` iterator and steps through it one value at a time.

Topics covered:

- `Sequence` — the C++ type that wraps a daslang generator
- `evalWithCatch` with a third `&Sequence` parameter to capture generators
- `builtin_iterator_iterate` — single-step the generator
- `builtin_iterator_close` — clean up generator resources

Prerequisites

- Tutorial 01 completed (tutorial_integration_cpp_hello_world).
- Familiarity with daslang generators (`generator<T>` and `yield`).

The daslang side

The script defines a function that returns a generator of integers. Each `yield` pauses execution and passes a value to the host:

```
options gen2

[export]
def test {
    return <- generator<int>() <| $() {
        for (i in range(5)) {
            print("  [das] yielding {i}\n")
            yield i
        }
        print("  [das] generator finished\n")
        return false
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The function creates the generator with `generator<int>() <| $()`, fills it with a loop, and transfers ownership to the caller with `return <-`.

Consuming a generator from C++

After compiling and simulating the script, we find the `test` function and capture its returned generator into a `Sequence`:

```
Sequence it;
ctx.evalWithCatch(fnTest, nullptr, &it);
```

When the third argument of `evalWithCatch` is a pointer to `Sequence`, the runtime fills it with the generator returned by the function.

Stepping through values

`builtin_iterator_iterate` advances the generator to its next `yield` and writes the yielded value into a caller-provided buffer:

```
int32_t value = 0;
int step = 0;
while (builtin_iterator_iterate(it, &value, &ctx)) {
    tout << "  [c++] step " << step << " => value " << value << "\n";
    step++;
}
```

Each call resumes the daslang generator, which runs until the next `yield` (or returns `false` to finish).

Cleanup

Always call `builtin_iterator_close` when done — even if the generator has already finished:

```
builtin_iterator_close(it, &value, &ctx);
```

This releases any resources held by the `Sequence`. If you break out of the iteration early (before the generator returns `false`), this call is essential.

Build & run

```
cmake --build build --config Release --target integration_cpp_17
bin/Release/integration_cpp_17
```

Expected output:

```
  [das] yielding 0
  [c++] step 0 => value 0
  [das] yielding 1
  [c++] step 1 => value 1
```

(continues on next page)

(continued from previous page)

```
  [das] yielding 2
  [c++] step 2 => value 2
  [das] yielding 3
  [c++] step 3 => value 3
  [das] yielding 4
  [c++] step 4 => value 4
  [das] generator finished
Generator produced 5 values total
```

The output is interleaved: each `yield` in daslang produces a `[das]` line, and the C++ iteration loop produces a `[c++]` line.

See also:

Full source: `17_coroutines.cpp`, `17_coroutines.das`

Previous tutorial: tutorial_integration_cpp_sandbox

Next tutorial: tutorial_integration_cpp_dynamic_scripts

Related: *Generator*

5.3.18 C++ Integration: Dynamic Scripts

This tutorial shows how to build a daslang program **from a string** at runtime, compile it from a **virtual file** (no disk I/O), and interact with its global variables directly through context memory pointers.

The example implements an expression calculator: the host defines variables and a math expression, compiles them into a tiny daslang program, and then evaluates the expression at near-native speed by poking variable values directly into context memory.

Topics covered:

- `TextWriter` — building daslang source code as a C++ string
- `TextFileInfo` + `FsFileAccess::setFileInfo` — registering a virtual file
- `compileDaScript` on a virtual file name
- `ctx.findVariable` / `ctx.getVariable` — locating globals in context memory
- Direct pointer writes for maximum-performance variable updates
- `shared_ptr<Context>` for long-lived context reuse

Prerequisites

- Tutorial 11 completed (tutorial_integration_cpp_context_variables).
- Understanding of `compileDaScript` and `Context` lifecycle.

Building source from C++ strings

`TextWriter` is daslang's stream builder. We use it to construct a valid daslang program with global variables and an eval function:

```
TextWriter ss;
ss << "options gen2\n";
ss << "require math\n\n";
for (auto & v : vars) {
    ss << "var " << v.first << " = 0.0f\n";
}
ss << "\n[export]\n"
   << "def eval() : float {\n"
   << "    return " << expr << "\n"
   << "}\n";
```

The generated source is a complete daslang module — it `requires` `math` for trigonometric functions and declares each variable as a module-level global.

Compiling from a virtual file

Instead of writing the source to disk, we register it as a virtual file using `TextFileInfo` and `setFileInfo`:

```
auto fAccess = make_smart<FsFileAccess>();
auto fileInfo = make_unique<TextFileInfo>(
    text.c_str(), uint32_t(text.length()), false);
fAccess->setFileInfo("expr.das", das::move(fileInfo));

ModuleGroup dummyLibGroup;
auto program = compileDaScript("expr.das", fAccess, tout, dummyLibGroup);
```

`compileDaScript` treats `"expr.das"` as a real file, but the file access layer resolves it to our in-memory buffer. This is useful for procedurally generated code, REPL implementations, and expression evaluators.

Direct context memory access

After simulation, we locate each variable in context memory and cache its pointer for fast writes:

```
for (auto & v : vars) {
    auto idx = ctx->findVariable(v.first.c_str());
    if (idx != -1) {
        variables[v.first] = (float *)ctx->getVariable(idx);
    }
}
```

To evaluate the expression with new values, we write directly into context memory — no function calls, no marshalling:

```
float ExprCalc::compute(ExprVars & vars) {
    for (auto & v : vars) {
        auto it = variables.find(v.first);
        if (it != variables.end()) {
            *(it->second) = v.second;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    auto res = ctx->evalWithCatch(fni, nullptr);
    return cast<float>::to(res);
}
```

Build & run

```
cmake --build build --config Release --target integration_cpp_18
bin/Release/integration_cpp_18
```

Expected output:

```
a=1 b=2 c=3  =>  a*b+c = 5
All 60 evaluations correct
sin(0)+cos(0) = 1
sin(pi)+cos(pi) = -1
```

See also:

Full source: `18_dynamic_scripts.cpp`

Previous tutorial: tutorial_integration_cpp_coroutines

Next tutorial: tutorial_integration_cpp_class_adapters

Related: tutorial_integration_cpp_context_variables

5.3.19 C++ Integration: Class Adapters

This tutorial shows how to let daslang classes **derive from a C++ abstract base class** and have C++ call their virtual methods seamlessly. This is the key pattern for game object systems where scripts define behavior but C++ owns the update loop.

Topics covered:

- The 3-layer adapter pattern (base → generated adapter → bridge)
- Pre-generated class adapter via `daslib/cpp_bind`
- `compileBuiltinModule` with XDD-embedded `.das.inc` files
- RTTI (`StructInfo *`) for adapter construction
- `getDasClassMethod` / `das_invoke_function` for virtual dispatch
- Virtual dispatch across the C++/daslang boundary

Architecture

The pattern uses three layers:

1. **C++ base class** (`BaseClass`) — pure virtual interface that C++ code programs against.
2. **Generated adapter** (`TutorialBaseClass`) — generated by `daslib/cpp_bind`'s `log_cpp_class_adapter`. For each virtual method, it provides `get_<method>` (checks if the daslang class overrides the method) and `invoke_<method>` (calls it via `das_invoke_function`).
3. **Dual-inheritance bridge** (`BaseClassAdapter`) — inherits both the real C++ class and the generated adapter. When C++ calls `obj->update(dt)` through the vtable, the bridge checks for a daslang override and invokes it.

Prerequisites

- Tutorial 08 (tutorial_integration_cpp_methods) — binding methods.
- Tutorial 10 (tutorial_integration_cpp_custom_modules) — custom modules.
- Understanding of `ManagedStructureAnnotation` and `das_invoke_function`.

Layer 1: C++ base class

A simple abstract interface that C++ code iterates over:

```
class BaseClass {
public:
    virtual ~BaseClass() = default;
    virtual void update(float dt) = 0;
    virtual float3 getPosition() = 0;
};
```

C++ maintains a list of `shared_ptr<BaseClass>` objects and calls `update` / `getPosition` polymorphically.

Layer 2: Generated adapter

The adapter is generated by `daslib/cpp_bind` (see `log_cpp_class_adapter`). It lives in a `.inc` file that you `#include` in your C++ source:

```
#include "19_class_adapters_gen.inc"
```

The generated code provides:

- `TutorialBaseClass` — stores `StructInfo` and method lookup data
- `get_update(classPtr)` — returns the `Func` if the daslang class overrides `update`, or `nullptr` if it does not
- `invoke_update(ctx, fn, classPtr, dt)` — calls the override via `das_invoke_function<void>::invoke`

Layer 3: Bridge class

The bridge inherits both `BaseClass` (for the vtable) and `TutorialBaseClass` (for daslang method lookup):

```
class BaseClassAdapter : public BaseClass,
                         public TutorialBaseClass {
public:
    BaseClassAdapter(char * pClass,
                     const StructInfo * info,
                     Context * ctx)
        : TutorialBaseClass(info),
          classPtr(pClass), context(ctx) {}

    void update(float dt) override {
        if (auto fn = get_update(classPtr)) {
            invoke_update(context, fn, classPtr, dt);
        }
    }

    float3 getPosition() override {
        if (auto fn = get_get_position(classPtr)) {
            return invoke_get_position(context, fn, classPtr);
        }
        return float3(0.0f);
    }

protected:
    void *    classPtr;
    Context * context;
};
```

Module with XDD-embedded daslang

The abstract base class definition lives in a `.das` file that is embedded into C++ using XDD (a CMake macro that converts files to byte arrays). The module loads it with `compileBuiltinModule`:

```
#include "class_adapters_module.das.inc"

class Module_Tutorial19 : public Module {
public:
    Module_Tutorial19() : Module("tutorial_19") {
        ModuleLibrary lib(this);
        lib.addBuiltInModule();
        addBuiltinDependency(lib, Module::require("rtti"));

        addExtern<DAS_BIND_FUN(addObject)>(*this, lib, "add_object",
            SideEffects::modifyExternal, "addObject");

        compileBuiltinModule("class_adapters_module.das",
            class_adapters_module_das,
            sizeof(class_adapters_module_das));
    }
```

(continues on next page)

(continued from previous page)

```
};

REGISTER_MODULE(Module_Tutorial19);
```

The daslang side

Scripts derive from the exposed abstract class and use `def override` to provide implementations:

```
options gen2
require tutorial_19
require rtti

class ExampleObject : TutorialBaseClass {
    position : float3
    speed : float

    def override update(dt : float) : void {
        position.x += speed * dt
    }

    def override get_position() : float3 {
        return position
    }
}

[export]
def test {
    var obj = new ExampleObject()
    obj.position = float3(0.0, 0.0, 0.0)
    obj.speed = 10.0

    unsafe {
        add_object(addr(*obj), class_info(*obj), this_context())
    }

    let avg = tick(0.5)
    print("After tick(0.5): avg position = ({avg.x}, {avg.y}, {avg.z})\n")
}
```

Build & run

```
cmake --build build --config Release --target integration_cpp_19
bin/Release/integration_cpp_19
```

Expected output:

```
After tick(0.5): avg position = (5, 0, 0)
```

See also:

Full source: `19_class_adapters.cpp`, `19_class_adapters.das`, `class_adapters_module.das`,

`19_class_adapters_gen.inc`

Previous tutorial: tutorial_integration_cpp_dynamic_scripts

Next tutorial: tutorial_integration_cpp_standalone_contexts

Related: tutorial_integration_cpp_methods, tutorial_integration_cpp_custom_modules

5.3.20 C++ Integration: Standalone Contexts

This tutorial shows how to use a **pre-compiled standalone context** — a daslang program baked entirely into C++ with zero runtime compilation overhead.

Normal daslang embedding compiles `.das` files at startup: `compileDaScript` → `simulate` → `evalWithCatch`. A standalone context eliminates the compile and simulate steps entirely. The AOT tool generates a `.das.h` and `.das.cpp` that contain all function code, type information, and initialization logic as C++ code.

Topics covered:

- `DAS_AOT_CTX` CMake macro — generating standalone context artifacts
- The generated `Standalone` class (extends `Context`)
- Direct method calls — `test()` is a C++ function, no lookup needed
- Zero startup cost — no parsing, no type checking, no simulation

Prerequisites

- Tutorial 13 completed (tutorial_integration_cpp_aot).
- Understanding of AOT compilation and `Context`.

When to use standalone contexts

Standalone contexts are ideal when:

- **Startup time matters** — embedded systems, command-line tools, game loading screens where even milliseconds count.
- **No file system access** — the script is burned into the binary.
- **Maximum performance** — all functions are AOT-compiled with no fallback to the interpreter.

The trade-off is that the script is fixed at build time.

The daslang script

A simple script that computes a sum:

```
options gen2

[export]
def test {
    var total = 0
    for (i in range(10)) {
        total += i
```

(continues on next page)

(continued from previous page)

```
    }
    print("Sum of 0..9 = {total}\n")
}
```

Build pipeline

The `DAS_AOT_CTX` CMake macro drives the generation:

```
SET(STANDALONE_CTX_GENERATED_SRC)
DAS_AOT_CTX("tutorials/integration/cpp/standalone_context.das"
            STANDALONE_CTX_GENERATED_SRC
            integration_cpp_20_dasAotStubStandalone daslang)

add_executable(integration_cpp_20
    20_standalone_context.cpp
    standalone_context.das
    ${STANDALONE_CTX_GENERATED_SRC})
TARGET_LINK_LIBRARIES(integration_cpp_20 libDaScript)
```

This runs `daslang` with the `-ctx` flag, which invokes `daslib/aot_standalone.das` to generate:

- `standalone_context.das.h` — declares the `Standalone` class
- `standalone_context.das.cpp` — implements all AOT functions and context initialization

These files are placed in `_standalone_ctx_generated/`.

The C++ host

The host code is remarkably simple — no module initialization, no compilation, no simulation:

```
#include "daScript/daScript.h"
#include "_standalone_ctx_generated/standalone_context.das.h"

using namespace das;

int main(int, char * []) {
    TextPrinter tout;
    tout << "Creating standalone context...\n";

    auto ctx = standalone_context::Standalone();

    tout << "Calling test():\n";
    ctx.test();

    tout << "Done.\n";
    return 0;
}
```

Key points:

- **No** `NEED_ALL_DEFAULT_MODULES` or `Module::Initialize` — the standalone context is entirely self-contained.

- `standalone_context::Standalone()` — the constructor sets up all functions, globals, and type info from pre-generated AOT data.
- `ctx.test()` — a direct C++ method call, not `findFunction` followed by `evalWithCatch`.

Generated namespace

The generated namespace is derived from the script file name:

- `standalone_context.das` → namespace `das::standalone_context`
- Class name is always `Standalone`
- Each `[export]` function becomes a method: `ctx.test()`, `ctx.compute(args...)`, etc.

Build & run

```
cmake --build build --config Release --target integration_cpp_20
bin/Release/integration_cpp_20
```

Expected output:

```
Creating standalone context (no runtime compilation)...
Calling test():
Sum of 0..9 = 45
Done.
```

See also:

Full source: `20_standalone_context.cpp`, `standalone_context.das`

Previous tutorial: tutorial_integration_cpp_class_adapters

Next tutorial: tutorial_integration_cpp_threading

Related: tutorial_integration_cpp_aot

5.3.21 C++ Integration: Threading

This tutorial demonstrates how to use daslang contexts and compilation pipelines across multiple threads in a C++ host application.

Topics covered:

- **Part A** — Running a compiled context on a worker thread
- **Part B** — Compiling a script from scratch on a worker thread
- `daScriptEnvironment::getBound()` / `setBound()` — thread-local environment binding
- `ReuseCacheGuard` — thread-local free-list cache management
- `ContextCategory::thread_clone` — context clone category for threads
- `shared_ptr<Context>` with `sharedPtrContext = true`
- `CodeOfPolicies::threadlock_context` — context mutex for threads
- Per-thread `Module::Initialize()` / `Module::Shutdown()`

Prerequisites

- Tutorial 1 completed (tutorial_integration_cpp_hello_world) — basic compile → simulate → eval cycle.
- Understanding of `Context` and `daScriptEnvironment` from tutorial_integration_cpp_standalone_contexts.

Why threading matters

daslang uses **thread-local storage** (TLS) for its environment object (`daScriptEnvironment`). This object holds the module registry, compilation state, and various global flags. Every thread that touches daslang API must have a valid environment bound.

There are two common patterns:

1. **Share the environment** — the worker thread binds the same environment object that the main thread uses. This works when the worker only *executes* (not compiles) and the main thread is idle.
2. **Independent environment** — the worker thread creates its own module registry and compilation pipeline from scratch. This is fully isolated and safe for concurrent compilation.

The daslang script

A simple script that computes the sum 0 + 1 + … + 99 = 4950:

```
options gen2

[export]
def compute() : int {
    var total = 0
    for (i in range(100)) {
        total += i
    }
    return total
}
```

Part A — Run on a worker thread

Compile and simulate on the main thread, then clone the context and run it on a `std::thread`.

```
// 1. Compile & simulate on main thread (standard boilerplate).
CodeOfPolicies policies;
policies.threadlock_context = true;  // context will run on another thread
auto program = compileDaScript(getDasRoot() + SCRIPT_NAME,
                               fAccess, tout, dummyLibGroup, policies);
Context ctx(program->getContextStackSize());
program->simulate(ctx, tout);
auto fnCompute = ctx.findFunction("compute");

// 2. Clone the context for the worker thread.
shared_ptr<Context> threadCtx;
threadCtx.reset(new Context(ctx, uint32_t(ContextCategory::thread_clone)));
threadCtx->sharedPtrContext = true;
```

(continues on next page)

(continued from previous page)

```
auto fnComputeClone = threadCtx->findFunction("compute");

// 3. Capture the current environment.
auto bound = daScriptEnvironment::getBound();

// 4. Launch the worker thread.
int32_t result = 0;
std::thread worker([&result, threadCtx, fnComputeClone, bound]() mutable {
    daScriptEnvironment::setBound(bound);
    vec4f res = threadCtx->evalWithCatch(fnComputeClone, nullptr);
    result = cast<int32_t>::to(res);
});
worker.join();
```

Key points:

Concept	Explanation
`daScriptEnvironment::getBound()`	Returns a pointer to the current thread's environment (TLS). Must be captured **before** launching the worker.
`daScriptEnvironment::setBound(env)`	Binds the environment on the worker thread. Without this, any daslang API call will crash.
`ContextCategory::thread_clone`	Marks the context as a thread-owned clone. The runtime can use this flag for diagnostics and thread-safety checks.
`shared_ptr<Context>`	Ensures the clone's lifetime extends until the worker is done. Set `sharedPtrContext = true` so the delete path is correct.
`threadlock_context`	Compile-time policy that gives the context a mutex. Required when the context (or its clone) will be accessed from a non-main thread.
`findFunction` on clone	Each context has its own function table. Always look up functions on the context that will execute them.

Part B — Compile on a worker thread

Create a fully independent daslang environment on a new thread. This is the pattern used in `test_threads.cpp` for concurrent compilation benchmarks.

```
std::thread worker([&result]() {
    // 1. Thread-local free-list cache.
    ReuseCacheGuard guard;

    // 2. Register module factories.
    NEED_ALL_DEFAULT_MODULES;

    // 3. Initialize modules - creates a fresh environment in TLS.
    Module::Initialize();

    // 4. Standard compile → simulate → eval cycle.
    TextPrinter tout;
    ModuleGroup dummyLibGroup;
    CodeOfPolicies policies;
    policies.threadlock_context = true;  // context mutex for thread safety
    auto fAccess = make_smart<FsFileAccess>();
```

(continues on next page)

(continued from previous page)

```
    auto program = compileDaScript(getDasRoot() + SCRIPT_NAME,
                                   fAccess, tout, dummyLibGroup, policies);
    Context ctx(program->getContextStackSize());
    program->simulate(ctx, tout);
    auto fn = ctx.findFunction("compute");
    vec4f res = ctx.evalWithCatch(fn, nullptr);
    result = cast<int32_t>::to(res);

    program.reset();

    // 5. Shut down this thread's modules.
    Module::Shutdown();
});
worker.join();
```

Key points:

Concept	Explanation
`ReuseCacheGuard`	Initializes and tears down per-thread free-list caches. Must be created **first** on any thread that uses daslang.
`NEED_ALL_DEFAULT_MODULES`	Registers module factory functions (not instances). This is a macro that must run before `Module::Initialize()`.
`Module::Initialize()`	Creates a new `daScriptEnvironment` in TLS and instantiates all registered modules.
`Module::Shutdown()`	Destroys this thread's modules and environment. Must be called before the thread exits.
`threadlock_context`	Same policy as Part A — ensures the context has a mutex even when compiled on the worker thread.
`program.reset()`	Release the program before `Module::Shutdown()` to avoid dangling references to destroyed modules.

Choosing the right pattern

Criteria	Part A (clone + run)	Part B (compile on thread)
Compilation cost	Zero on worker (pre-compiled)	Full compilation on worker
Isolation	Shares environment with main	Fully independent
Concurrent compilation	Not safe (shared env)	Safe (separate env per thread)
Use case	Game threads running pre-compiled AI/logic	Build servers, parallel test runners

Build & run

```
cmake --build build --config Release --target integration_cpp_21
bin/Release/integration_cpp_21
```

Expected output:

```
=== Part A: Run on a worker thread ===
  compute() on worker thread returned: 4950
  PASS

=== Part B: Compile on a worker thread ===
  compute() compiled + run on worker thread returned: 4950
  PASS
```

See also:

Full source: `21_threading.cpp`, `21_threading.das`

Previous tutorial: tutorial_integration_cpp_standalone_contexts

Related: tutorial_integration_cpp_hello_world (basic compile/simulate/eval)

5.4 Macro Tutorials

These tutorials teach daslang's compile-time macro system: call macros, reader macros, function macros, and AST manipulation. Each tutorial has **two** source files — a module (`.das`) that defines the macros and a usage file that exercises them — because macros cannot be used in the same module that defines them.

Run any tutorial from the project root:

```
daslang.exe tutorials/macros/01_call_macro.das
```

5.4.1 Macro Tutorial 1: Call Macros

Call macros intercept function-call syntax at compile time and replace it with arbitrary AST. When the compiler sees `hello()` or `printf("...", args)`, it invokes your macro's `visit` method instead of looking for a function — giving you full control over what code is generated.

This tutorial builds three progressively complex call macros:

1. `hello()` — the simplest possible macro (no arguments)
2. `greet("name")` — argument validation and string builder construction
3. `printf(fmt, args...)` — format-string parsing with argument reordering

Note: Macros cannot be used in the module that defines them. Every macro tutorial therefore has **two** source files: a *module* file containing the macro definitions and a *usage* file that requires the module and exercises the macros.

Prerequisites

Familiarity with daslang basics (functions, strings, control flow) is assumed. No prior macro experience is required — concepts are introduced one at a time.

Key imports used by the module:

```
require daslib/ast              // AST node types (ExprConstString, etc.)
require daslib/ast_boost        // AST helpers and ExpressionPtr
require daslib/templates_boost  // qmacro, $e() reification
require daslib/strings_boost    // ExprStringBuilder
require daslib/macro_boost      // [call_macro] annotation, macro_verify
```

Section 1 — hello(): Minimal call macro

A call macro is a class that extends `AstCallMacro`, annotated with `[call_macro(name="...")]`:

```
[call_macro(name="hello")]
class HelloMacro : AstCallMacro {
    def override visit(prog : ProgramPtr; mod : Module?;
            var expr : smart_ptr<ExprCallMacro>) : ExpressionPtr {
        macro_verify(length(expr.arguments) == 0, prog, expr.at,
            "hello() takes no arguments")
        return <- qmacro(print("hello, call macro!\n"))
    }
}
```

The `visit` method receives:

- **prog** — the program being compiled (used for error reporting)
- **mod** — the module where the call appears
- **expr** — the call expression (with `.arguments` and `.at` for source location)

It returns an `ExpressionPtr` — the AST tree that replaces the call. `qmacro(...)` is a *reification* helper: you write normal daslang syntax inside it and it builds the corresponding AST at compile time.

Usage:

```
hello()   // → print("hello, call macro!\n")
```

Section 2 — greet("name"): Argument validation

The `greet` macro validates its single argument and builds a string interpolation expression:

```
[call_macro(name="greet")]
class GreetMacro : AstCallMacro {
    def override visit(prog : ProgramPtr; mod : Module?;
            var expr : smart_ptr<ExprCallMacro>) : ExpressionPtr {
        macro_verify(length(expr.arguments) == 1, prog, expr.at,
            "greet() requires exactly one string argument")
        macro_verify(expr.arguments[0] is ExprConstString, prog, expr.at,
            "greet() argument must be a string literal")
```

(continues on next page)

(continued from previous page)

```
        var inscope sbuilder <- new ExprStringBuilder(at = expr.at)
        sbuilder.elements |> emplace_new <| new ExprConstString(
            value := "hello, ", at = expr.at)
        sbuilder.elements |> emplace_new <| clone_expression(expr.arguments[0])
        sbuilder.elements |> emplace_new <| new ExprConstString(
            value := "!\n", at = expr.at)
        return <- qmacro(print($e(sbuilder)))
    }
}
```

Key techniques:

- ``**expr.arguments[0] is ExprConstString**`` — compile-time type check on the AST node to verify the argument is a string literal.
- ``**macro_verify**`` — emits a compile error and returns an empty expression if the condition is false.
- ``**ExprStringBuilder**`` — the AST node for string interpolation (`"hello, {name}!\n"`). Its `.elements` array holds literal strings and interpolated expressions.
- ``**clone_expression**`` — duplicates an AST node. Always clone arguments before inserting them into new AST — the original may be used elsewhere.
- ``**$e(expr)**`` inside `qmacro` — splices an expression node into the reified AST.

Usage:

```
greet("world")    // → print("hello, world!\n")
greet("daslang")  // → print("hello, daslang!\n")
```

Section 3 — printf(fmt, args...): Format-string parsing

The `printf` macro parses a format string at compile time, replacing `(N)` placeholders with the corresponding argument expressions:

```
printf("player (1) scored (2) points\n", "Alice", score)
// → print("player {\"Alice\"} scored {score} points\n")
```

Arguments can be **reordered** and **repeated**:

```
printf("result: (2) from (1)\n", "source", 100)
printf("(1) and (1) and (1)\n", "echo")
```

The implementation iterates over the format string character by character, looking for `(...)` pairs. For each placeholder it:

1. Extracts the number with `chop` and converts it with `to_int`
2. Validates bounds with `macro_verify`
3. Inserts a `clone_expression` of the referenced argument

```
[call_macro(name="printf")]
class PrintfMacro : AstCallMacro {
    def override visit(prog : ProgramPtr; mod : Module?;
            var expr : smart_ptr<ExprCallMacro>) : ExpressionPtr {
```

(continues on next page)

(continued from previous page)

```
        macro_verify(length(expr.arguments) >= 1, prog, expr.at,
            "printf requires at least a format string argument")
        macro_verify(expr.arguments[0] is ExprConstString, prog, expr.at,
            "first argument to printf must be a constant string")
        let totalArgs = length(expr.arguments)
        var inscope sbuilder <- new ExprStringBuilder(at = expr.at)
        let format = string((expr.arguments[0] as ExprConstString).value)
        var pos = 0
        while (pos < length(format)) {
            var open = find(format, '(', pos)
            if (open == -1) {
                let tail = format.chop(pos, length(format) - pos)
                sbuilder.elements |> emplace_new <| new ExprConstString(
                    value := tail, at = expr.at)
                break
            }
            if (open > pos) {
                let text = format.chop(pos, open - pos)
                sbuilder.elements |> emplace_new <| new ExprConstString(
                    value := text, at = expr.at)
            }
            var close = find(format, ')', open + 1)
            macro_verify(close != -1, prog, expr.at,
                "unmatched '(' in format string")
            var argNumStr = format.chop(open + 1, close - open - 1)
            var argNum = to_int(argNumStr)
            macro_verify(argNum >= 1, prog, expr.at,
                "argument number must be >= 1")
            macro_verify(argNum < totalArgs, prog, expr.at,
                "argument index out of range")
            sbuilder.elements |> emplace_new <| clone_expression(
                expr.arguments[argNum])
            pos = close + 1
        }
        return <- qmacro(print($e(sbuilder)))
    }
}
```

Running the tutorial

```
daslang.exe tutorials/macros/01_call_macro.das
```

Expected output:

```
hello, call macro!
hello, world!
hello, daslang!
player Alice scored 42 points
result: 100 from source
echo and echo and echo
pi is approximately 3.14, or roughly 3
```

See also:

Full source: `call_macro_mod.das`, `01_call_macro.das`

Next tutorial: tutorial_macro_when_expression

Language reference: *Macros* — full macro system documentation

5.4.2 Macro Tutorial 2: When Expression

This tutorial builds a `when` call macro — a value-returning match expression that compiles `=>` tuples inside a block into a chain of equality tests. It introduces several advanced techniques beyond the basics covered in tutorial_macro_call_macro.

```
let result = when(x) {
    1 => "one"
    2 => "two"
    _ => "other"
}
```

The macro transforms this into:

```
let result = invoke($(arg : int const) {
    if (arg == 1) { return "one"; }
    if (arg == 2) { return "two"; }
    return "other";
}, x)
```

New concepts introduced:

- ``**canVisitArgument**`` — control argument type-checking order
- ``**canFoldReturnResult**`` — defer return-type inference
- ``**qmacro_block**`` — block-level (statement list) reification
- ``**$i()**``, ``**$t()**``, ``**$b()**`` — inject identifiers, types, and lists
- ``**clone_type**`` and type flag manipulation
- ``**can_shadow**`` flag for generated variables
- ``**typedecl($e(...))**`` — infer types from expression nodes
- ``**default<T>**`` — generate type-safe default values

Prerequisites

You should be comfortable with the material in tutorial_macro_call_macro — `[call_macro]`, `visit()`, `macro_verify`, `qmacro`, and `$e()`.

How macro_verify works

Before diving in, an important detail: `macro_verify` is not a simple function. It is a *tag function macro* (defined in `daslib/macro_boost`) that expands to:

```
if (!condition) {
    macro_error(prog, at, message)
    return <- default<ExpressionPtr>
}
```

This means `macro_verify` **short-circuits** — if the condition is false, the enclosing `visit()` returns an empty expression immediately. Code after `macro_verify` only runs when the condition passed.

Controlling argument visitation

A call macro's arguments are type-checked by the compiler *before* `visit()` is called. Sometimes you need to control this — for example, the `when` block contains => tuples that should be parsed but not fully error-checked until after the macro transforms them.

`canVisitArgument` lets you decide per-argument:

```
def override canVisitArgument(expr : smart_ptr<ExprCallMacro>;
        argIndex : int) : bool {
    return true if (argIndex == 0)
    return !is_reporting_compilation_errors()
}
```

- **Argument 0** (the condition value): always visited, so its `_type` is available in `visit()`.
- **Argument 1** (the block): visited during normal compilation (`is_reporting_compilation_errors()` is false → returns true) but **not** during the error-reporting pass (after the macro has already transformed it).

Deferring return-type inference

`canFoldReturnResult` tells the compiler whether the return type of the enclosing function can be finalized while this macro call is still unexpanded:

```
def override canFoldReturnResult(
        expr : smart_ptr<ExprCallMacro>) : bool {
    return false
}
```

Returning `false` prevents the compiler from concluding "this function returns void" before `when` has a chance to produce its `invoke()` expression.

Building the statement list

The core of the macro iterates over the block's statements. Each statement is an `ExprMakeTuple` (the => operator creates tuples):

```
for (stmt, idx in blk.list, count()) {
    let tupl = stmt as ExprMakeTuple
    assume cond_value = tupl.values[0]
    let is_default = (cond_value is ExprVar)
                  && (cond_value as ExprVar).name == "_"
```

For each case, we build either a conditional return or an unconditional return using `qmacro_block`:

```
if (is_default) {
    list |> emplace_new <| qmacro_block() {
        return $e(tupl.values[1])
    }
} else {
    list |> emplace_new <| qmacro_block() {
        if ($i(arg_name) == $e(tupl.values[0])) {
            return $e(tupl.values[1])
        }
    }
}
```

Key reification escapes:

- **``$e(expr)``** — splices an expression node (as in tutorial 1)
- **``$i(name)``** — converts a string into an identifier (`ExprVar`)
- **``qmacro_block() { ... }``** — produces a statement list (`ExpressionPtr`) rather than a single expression

Assembling the block

After building the statement list, we need a typed block argument. `clone_type` copies the condition's inferred type, and we adjust flags:

```
var inscope cond_type <- clone_type(cond._type)
cond_type.flags.ref = false      // compare values, not references
cond_type.flags.constant = true  // argument is read-only
```

Then we assemble the block and mark its argument as shadowable:

```
var inscope call_block <- qmacro(
    $($i(arg_name) : $t(cond_type)){ $b(list); })
((call_block as ExprMakeBlock)._block as ExprBlock)
    .arguments[0].flags.can_shadow = true
```

- **``$t(type)``** — injects a `TypeDeclPtr` into the reified AST
- **``$b(list)``** — injects an `array<ExpressionPtr>` as the block body
- **``can_shadow``** — allows nested `when()` calls to each introduce their own `__when_arg__` without name conflicts

The final result is an `invoke` call:

```
return <- qmacro(invoke($e(call_block), $e(cond)))
```

Auto-generated default case

If no _ default case is provided, the generated block would have no return on some code paths — the compiler would reject it. The macro solves this by automatically generating a default that returns the type's default value ("" for strings, 0 for ints, etc.):

```
if (!any_default) {
    assume first_value = (blk.list[0] as ExprMakeTuple).values[1]
    list |> emplace_new <| qmacro_block() {
        return default<typedecl($e(first_value))>
    }
}
```

- ``**typedecl($e(expr))**`` — extracts the type from an expression node. Here we use the first case's value expression to infer what type the `when` block should return.
- ``**default<T>**`` — produces the default value for type `T` (empty string, zero, null pointer, etc.).

This means `when()` without _ is safe:

```
let found = when(y) {
    1 => "found one"
    2 => "found two"
}
// if y is neither 1 nor 2, found == "" (default string)
```

Usage examples

Integer matching:

```
var x = 2
let result = when(x) {
    1 => "one"
    2 => "two"
    _ => "other"
}
// result == "two"
```

String matching:

```
let lang = "daslang"
let greeting = when(lang) {
    "python"  => "import this"
    "daslang" => "hello, call macro!"
    _         => "unknown language"
}
```

Nested when (`can_shadow` in action):

```
let nested = when(a) {
    1 => when(b) {
        1 => "a=1, b=1"
        2 => "a=1, b=2"
        _ => "a=1, b=other"
    }
    _ => "a=other"
}
```

Running the tutorial

```
daslang.exe tutorials/macros/02_when_macro.das
```

Expected output:

```
x=2: two
lang=daslang: hello, call macro!
n=3: triple (3 items)
y=42: ''
nested: a=1, b=2
val=3.14: pi
```

See also:

Full source: `when_macro_mod.das`, `02_when_macro.das`

Previous tutorial: tutorial_macro_call_macro

Next tutorial: tutorial_macro_function_macro

Language reference: *Macros* — full macro system documentation

5.4.3 Macro Tutorial 3: Function Macros

This tutorial builds three function macros that demonstrate the three main methods of `AstFunctionAnnotation`:

- ``**[log_calls]**`` uses `apply()` to rewrite a function's body, adding entry/exit logging with nested-call indentation.
- ``**[expect_range]**`` uses `verifyCall()` to validate every call site, rejecting constant arguments that fall outside a given range.
- ``**[no_print]**`` uses `lint()` to walk the fully-compiled body and reject calls to the builtin `print` function.

```
[log_calls]
def add(a, b : int) : int {
    return a + b
}
```

At compile time, `[log_calls]` rewrites the function body to:

```
def add(a, b : int) : int {
    if (true) {
        print(">> ")
        print("add({a}, {b})\n")
```

(continues on next page)

(continued from previous page)

```
        if (true) {
            return a + b        // original body
        }
    } finally {
        print("<< add\n")
    }
}
```

Recursive calls produce indented output that visualizes the call tree:

```
>> fib(3)
  >> fib(2)
    >> fib(1)
    << fib
    >> fib(0)
    << fib
  << fib
  >> fib(1)
  << fib
<< fib
```

New concepts introduced:

- ``**[function_macro]**`` — macro kind that modifies functions
- ``**AstFunctionAnnotation**`` — base class with `apply()`, `verifyCall()`, and `lint()` methods
- ``**apply**()`` — transforms a function's AST at definition time
- ``**verifyCall**()`` — validates each call site after type inference
- ``**lint**()`` — walks the fully-compiled AST after all types are resolved
- ``**ExprStringBuilder**`` — build string interpolations from AST nodes
- ``**qmacro_block**`` **with** ``**finally**`` — generate statement blocks with cleanup sections
- ``**if (true) { ... }**`` — scoping trick for variable isolation
- ``**$e(func.body)**`` — inject the original function body
- ``**func.body |> move**`` — replace the function body
- ``**var public**`` — module-level mutable state shared across modules
- ``**AnnotationArgumentList**`` — reading annotation argument names and values (`iValue`)
- ``**ExprConstInt**`` — extracting compile-time integer values from AST
- ``**AstVisitor**`` — walking the compiled AST tree
- ``**make_visitor**`` / ``**visit**()`` — adapting and running a visitor
- ``**expr.func._module.name**`` — identifying a function's source module

Prerequisites

You should be comfortable with the material in tutorial_macro_call_macro and tutorial_macro_when_expression — `[call_macro]`, `visit()`, `qmacro`, `$e()`, `$i()`, and `qmacro_block`.

Function macros vs. call macros

Call macros (tutorials 1 and 2) transform **call expressions** — they receive a call site and return a replacement expression.

Function macros transform **function definitions**. They receive a `FunctionPtr` and modify the function's AST (body, arguments, return type, annotations) before the function is compiled. The base class is `AstFunctionAnnotation` and it provides several overridable methods:

- ``**apply**()`` — runs once when the function is compiled. Use it to transform the function's body, arguments, or annotations.
- ``**verifyCall**()`` — runs at every call site after type inference. Use it to validate arguments (return `false` to reject the call).
- ``**transform**()`` — runs at every call site and can replace the call expression entirely (used by `[constant_expression]` in daslib).
- ``**lint**()`` — runs after the function is fully compiled (types resolved, overloads selected). Use it to validate structural properties of the finished AST.

This tutorial demonstrates `apply()` with `[log_calls]`, `verifyCall()` with `[expect_range]`, and `lint()` with `[no_print]`.

Part 1: [log_calls] — apply()

The `apply()` method receives the function being compiled and can modify its AST arbitrarily:

```
[function_macro(name="log_calls")]
class LogCallsMacro : AstFunctionAnnotation {
    def override apply(var func : FunctionPtr;
                       var group : ModuleGroup;
                       args : AnnotationArgumentList;
                       var errors : das_string) : bool {
        // ... transform func ...
        return true
    }
}
```

Returning `true` from `apply()` means the transformation succeeded. Returning `false` aborts compilation with an error.

Building the call signature string

To log which function was called and with what arguments, we need a string like `"add(2, 3)\n"` at runtime. This must be built as an `ExprStringBuilder` — a compile-time AST node that generates string interpolation code:

```
var inscope call_sb <- new ExprStringBuilder(at = func.at)
call_sb.elements |> emplace_new <| qmacro($v("{string(func.name)}("))
for (i, arg in count(), func.arguments) {
    if (i > 0) {
        call_sb.elements |> emplace_new <| quote(", ")
    }
    call_sb.elements |> emplace_new <| qmacro($i(arg.name))
}
call_sb.elements |> emplace_new <| quote(")\n")
```

How it works:

- ``**ExprStringBuilder**`` is the AST node behind daslang's `"..."` string interpolation. Its `elements` array holds a mix of literal strings and expression nodes that become `{expr}` segments.
- ``**$v("text")**`` (value) injects a constant string — here the function name and opening parenthesis.
- ``**$i(arg.name)**`` (identifier) creates a variable reference — at runtime it evaluates to the argument's actual value.
- ``**quote("text")**`` creates a literal string expression — used for fixed separators like `", "` and `")\n"`.
- ``**count()**`` and ``**func.arguments**`` — the arguments array on `FunctionPtr` holds the function's parameter declarations. `count()` provides a 0-based index for the comma-separator logic.

For `add(a, b : int)`, this produces the equivalent of:

```
"{string(func.name)}({a}, {b})\n"
```

which at runtime evaluates to `"add(2, 3)\n"`.

The if(true) scoping pattern

The generated code uses `if (true) { ... }` blocks that may look redundant. They serve a real purpose — each `if (true) { ... }` creates a **new lexical scope**. This is important because:

1. The macro may introduce local variables (like `ref_time` in extended versions). Scoping prevents name clashes with the original body.
2. The original body may contain `return` statements. Wrapping it in a scope ensures `finally` still runs.
3. Multiple `[log_calls]` annotations (or other function macros) can each add their own scoped variables without conflicts.

Constructing the replacement body

The heart of the macro builds a new function body using `qmacro_block`. This generates a statement list (`ExprBlock`) rather than a single expression:

```
var inscope new_body <- qmacro_block() {
    if (true) {
        print("{repeat("  ",LOG_DEPTH++)}>> ")
        print($e(call_sb))
        if (true) {
            $e(func.body)
        }
    } finally {
        print("{repeat("  ",--LOG_DEPTH)}<< {$v(string(func.name))}\n")
    }
}
```

Key details:

- **``qmacro_block() { ... }``** produces an `ExpressionPtr` containing a block of statements (an `ExprBlock`). Unlike `qmacro()` which produces a single expression, `qmacro_block` can hold multiple statements, `if`/`else`, `finally`, etc.
- **``$e(call_sb)``** splices in the `ExprStringBuilder` we built earlier. At runtime this becomes the `print("add(2, 3)\n")` call.
- **``$e(func.body)``** splices the function's **original body** into the inner `if (true)` block. This is the key technique — the macro wraps the original code rather than replacing it.
- **``finally { ... }``** — the generated block has a `finally` section that runs even when the original body executes a `return`. This guarantees the exit log line is always printed and `LOG_DEPTH` is decremented.
- **``LOG_DEPTH++``** / **``--LOG_DEPTH``** — pre/post-increment controls indentation depth. `repeat("  ", LOG_DEPTH++)` prints the current depth's indentation then increments; `--LOG_DEPTH` decrements before printing the exit indentation.
- **``$v(string(func.name))``** injects the function name as a compile-time constant string into the exit log.

Replacing the function body

Finally, we swap the function's body with our new block:

```
func.body |> move <| new_body
return true
```

`move` replaces `func.body` with `new_body` and clears `new_body`. This is the standard pattern for function body replacement in `apply()` — the old body has already been captured inside the new one via `$e(func.body)`, so no information is lost.

Public variables for shared state

`LOG_DEPTH` is declared at module scope with `var public`:

```
var public LOG_DEPTH = 0
```

This makes it accessible from any module that `require`-s `function_macro_mod`. Each `[log_calls]` function increments it on entry and decrements on exit, producing correct indentation for nested calls. Because it is a single global variable, it tracks depth across all annotated functions — not just recursive ones.

Part 2: [expect_range] — verifyCall()

While `apply()` transforms the function at definition time, `verifyCall()` runs at every **call site** after type inference. It receives the call expression and can accept or reject it.

```
[function_macro(name="expect_range")]
class ExpectRangeMacro : AstFunctionAnnotation {
    def override verifyCall(var call : smart_ptr<ExprCallFunc>;
                            args, progArgs : AnnotationArgumentList;
                            var errors : das_string) : bool {
        // ... validate call.arguments ...
        return true
    }
}
```

The parameters:

- ``**call**`` — the call expression at the call site. `call.func` is the function being called, `call.arguments` are the argument expressions.
- ``**args**`` — the annotation's argument list (e.g., for `[expect_range(value, min=0, max=255)]`, it contains three entries).
- ``**errors**`` — an output string for the error message. Set it and return `false` to produce a compile error.

Returning `false` emits error code `40102` (`annotation_failed`).

Reading annotation argument values

Annotation arguments like `[expect_range(value, min=0, max=255)]` are stored in an `AnnotationArgumentList`. Each entry has a `name` and typed value fields:

```
var arg_name = ""
var range_min = int(0x80000000)
var range_max = int(0x7FFFFFFF)
for (aa in args) {
    if (aa.basicType == Type.tBool) {
        arg_name = string(aa.name)   // bare name: "value"
    } elif (aa.name == "min") {
        range_min = aa.iValue         // integer value: 0
    } elif (aa.name == "max") {
        range_max = aa.iValue         // integer value: 255
    }
}
```

How it works:

- **Bare names** like `value` in `[expect_range(value, ...)]` are stored with `basicType == Type.tBool` and their name is the argument identifier. This is how `[constexpr(a)]` in daslib's `constant_expression.das` identifies which function parameter to check.
- **Named values** like `min=0` are stored with their name (`"min"`) and the integer value in `iValue`.
- ``**string(aa.name)**`` — converts the name for comparison. For `name == "min"` the comparison works directly.

Extracting constant integer values

To check whether a call-site argument is a compile-time constant and extract its value, we need a helper that navigates the AST:

```
[macro_function]
def public getConstantInt(expr : ExpressionPtr;
                          var result : int&) : bool {
    if (expr is ExprRef2Value) {
        return getConstantInt(
            (expr as ExprRef2Value).subexpr, result)
    } elif (expr is ExprConstInt) {
        result = (expr as ExprConstInt).value
        return true
    }
    return false
}
```

Key details:

- ``**ExprRef2Value**`` — the compiler sometimes wraps constant values in a reference-to-value conversion node. The helper unwraps it recursively via `.subexpr`.
- ``**ExprConstInt**`` — one of the `ExprConst*` family of AST nodes (`ExprConstFloat`, `ExprConstString`, `ExprConstBool`, etc.). All have a `.value` field of the corresponding type.
- ``**is**`` / ``**as**`` — daslang's type-test and downcast operators work on AST node types just like on classes.
- ``**[macro_function]**`` — marks the function as available during compilation (macro expansion time). Without this annotation, the function would only exist at runtime.

Daslib's `constant_expression.das` uses a more general approach: `expr.__rtti |> starts_with("ExprConst")` checks for *any* constant type. Our helper is specific to integers because `[expect_range]` only makes sense for numeric bounds.

Reporting compile-time errors

The error reporting pattern is straightforward — set the `errors` string and return `false`:

```
var val = 0
if (getConstantInt(ce, val)) {
    if (val < range_min || val > range_max) {
        errors := "{arg_name} = {val} is out of range [{range_min}..{range_max}]"
        return false
    }
}
```

The compiler wraps this into a full error message:

```
error[40102]: call annotated by expect_range failed
_test_error.das:12:4
    set_channel("red", 300)
    ^^^^^^^^^^^
value = 300 is out of range [0..255]
```

The error code `40102` (`annotation_failed`) is always the same for `verifyCall` failures. The string you set in `errors` becomes the detail message below the source location.

Runtime values pass through

An important design decision: `verifyCall` only checks **constant** arguments. When a runtime variable is passed, `getConstantInt` returns `false` and the call is allowed:

```
var alpha = 200
set_channel("alpha", alpha)   // runtime value - compiles fine
```

This is intentional. `verifyCall` is a **best-effort compile-time check** — it catches mistakes in literal arguments but cannot validate runtime expressions. For runtime validation, you would use `apply()` to inject runtime bounds checks into the function body.

Part 3: [no_print] — lint()

The `lint()` method runs after the function is **fully compiled** — types are resolved, overloads are selected, and the AST is ready to simulate. This makes it ideal for structural validation that needs complete type information.

```
[function_macro(name="no_print")]
class NoPrintMacro : AstFunctionAnnotation {
    def override lint(var func : FunctionPtr;
                      var group : ModuleGroup;
                      args, progArgs : AnnotationArgumentList;
                      var errors : das_string) : bool {
        // ... walk func body ...
        return true
    }
}
```

Compared to the other methods:

- ``**apply**()`` — runs at definition time, before type checking. The body has parsed expressions but types may not be resolved yet.
- ``**verifyCall**()`` — runs at each call site after type inference. Has access to call arguments but not the full function body.
- ``**lint**()`` — runs after everything is compiled. The function's `body` has full type annotations, `ExprCall` nodes have their `.func` pointers linked to the resolved `Function` objects.

Ironic contrast: `[log_calls]` *adds* print calls to every function, while `[no_print]` *forbids* them.

Walking the AST with a visitor

To inspect the function body, `lint()` uses the **visitor pattern**. We define a class that inherits from `AstVisitor` and overrides `preVisitExprCall` to intercept function calls:

```
[macro]
class NoPrintVisitor : AstVisitor {
    found_print : bool = false
    @safe_when_uninitialized print_at : LineInfo
    def override preVisitExprCall(
            expr : smart_ptr<ExprCall>) : void {
        if (expr.func != null
                && expr.name == "print"
                && expr.func._module.name == "$") {
            found_print = true
            print_at = expr.at
        }
    }
}
```

Key details:

- ``**[macro]**`` — required annotation for classes used during compilation. Without it, the visitor class would not exist at macro expansion time.
- ``**preVisitExprCall**`` — called before each `ExprCall` node in the AST walk. The `AstVisitor` base class has `preVisit*` and `visit*` hooks for every AST node type (`ExprFor`, `ExprWhile`, `ExprNew`, etc.).
- ``**@safe_when_uninitialized**`` — `LineInfo` is a struct with no default initializer. This annotation tells the compiler it is intentionally left uninitialized until `found_print` is set.
- ``**expr.func**`` — at lint time, this pointer is always linked to the resolved `Function` object (unlike at `apply()` time where function resolution may not be complete).

Checking the function's module

The check `expr.func._module.name == "$"` distinguishes the builtin `print` from any user-defined function that happens to be named `print`:

- ``**_module**`` — the field name uses an underscore prefix because `module` is a reserved keyword in daslang. In C++ the field is `Function::module`; in daslang macros it is `func._module`.
- ``**"$"**`` — the builtin module name. All built-in functions (`print`, `assert`, `length`, math functions, etc.) belong to module `"$"`.
- This pattern is used throughout daslib — for example, `daslib/lint.das` checks `expr.func._module.name |> eq <| "$"` to detect calls to `panic`.

Running the visitor from lint()

The `lint()` method creates the visitor, adapts it with `make_visitor`, and walks the function body with `visit()`:

```
def override lint(...) : bool {
    var astVisitor = new NoPrintVisitor()
    var inscope adapter <- make_visitor(*astVisitor)
    visit(func, adapter)
    if (astVisitor.found_print) {
        errors := "function {string(func.name)} must not call builtin print"
        unsafe { delete astVisitor; }
        return false
    }
    unsafe { delete astVisitor; }
    return true
}
```

- **``make_visitor(*astVisitor)``** — wraps the daslang visitor object into an adapter that the C++ `visit()` function can call. The `*` dereferences the smart pointer.
- **``visit(func, adapter)``** — walks the function's AST, calling the visitor's `preVisit*` / `visit*` hooks at each node.
- **Error reporting** uses the same pattern as `verifyCall()` — set `errors` and return `false`. The compiler emits error code `40102` with message text "`function annotation lint failed`" plus your detail string.
- **``unsafe { delete astVisitor; }``** — explicit cleanup of the heap-allocated visitor. Required because `new` allocates on the heap and the visitor is not managed by `inscope`.

Usage examples

[log_calls] — simple functions:

```
[log_calls]
def add(a, b : int) : int {
    return a + b
}

[log_calls]
def greet(name : string) {
    print("hello, {name}!\n")
}
```

Calling `add(2, 3)` produces:

```
>> add(2, 3)
<< add
```

Calling `greet("daslang")` produces:

```
>> greet(daslang)
hello, daslang!
<< greet
```

Recursive functions show the call tree via indentation:

```
[log_calls]
def fib(n : int) : int {
    if (n <= 1) {
        return n
    } else {
        return fib(n - 1) + fib(n - 2)
    }
}
```

Calling `fib(3)` produces:

```
>> fib(3)
  >> fib(2)
    >> fib(1)
    << fib
    >> fib(0)
    << fib
  << fib
  >> fib(1)
  << fib
<< fib
```

[expect_range] — compile-time bounds checking:

```
[expect_range(value, min=0, max=255)]
def set_channel(name : string; value : int) {
    print("  {name} = {value}\n")
}
```

Valid calls compile normally:

```
set_channel("red", 128)    // ok
set_channel("green", 0)    // ok
set_channel("blue", 255)   // ok
```

Out-of-range constants are rejected at compile time:

```
// set_channel("red", 300)   // compile error!
// error[40102]: call annotated by expect_range failed
//   value = 300 is out of range [0..255]
```

Runtime variables are allowed through:

```
var alpha = 200
set_channel("alpha", alpha)   // runtime value - passes through
```

[no_print] — lint-time structural validation:

```
[no_print]
def compute(a, b : int) : int {
    return a * b + 1
}
```

This compiles — `compute` has no `print` calls. Adding `[no_print]` to a function that calls `print` fails at lint time:

```
// [no_print]   // uncomment to see:
// error[40102]: function annotation lint failed
//   function bad_compute must not call builtin print
def bad_compute(a, b : int) : int {
    print("computing {a} * {b}\n")
    return a * b
}
```

Extending with timing

A natural extension is to add execution timing using `ref_time_ticks()` and `get_time_nsec()`. The pattern is the same — the only addition is a local timing variable in the generated body:

```
var inscope new_body <- qmacro_block() {
    if (true) {
        let ref_time = ref_time_ticks()
        print(">> ")
        print($e(call_sb))
        if (true) {
            $e(func.body)
        }
    } finally {
        print("<< {$v(string(func.name))} - {get_time_nsec(ref_time)}ns\n")
    }
}
```

The `if (true)` scope keeps `ref_time` local to each instrumented function, preventing name clashes when multiple `[log_calls]` functions call each other.

Running the tutorial

```
daslang.exe tutorials/macros/03_function_macro.das
```

Expected output:

```
>> add(2, 3)
<< add
sum = 5

>> greet(daslang)
hello, daslang!
<< greet

>> fib(3)
  >> fib(2)
    >> fib(1)
    << fib
    >> fib(0)
    << fib
  << fib
  >> fib(1)
```

(continues on next page)

(continued from previous page)

```
  << fib
<< fib
fib(3) = 2

color channels:
  red = 128
  green = 0
  blue = 255
  alpha = 200

compute = 13
```

See also:

Full source: `function_macro_mod.das`, `03_function_macro.das`

Previous tutorial: tutorial_macro_when_expression

Next tutorial: tutorial_macro_advanced_function_macro

Language reference: *Macros* — full macro system documentation

5.4.4 Macro Tutorial 4: Advanced Function Macros

This tutorial builds a `[memoize]` function macro that demonstrates the full `apply()` / `patch()` / `transform()` lifecycle of `AstFunctionAnnotation`. Where tutorial 3 used `apply()` alone to rewrite a function body, here we generate *new* companion functions, module-level cache variables, and redirect every call site — including recursive ones — to a memoized wrapper.

```
[memoize]
def fib(n : int) : int {
    if (n <= 1) { return n; }
    return fib(n - 1) + fib(n - 2)
}
```

Without memoization, `fib(30)` would take over a billion recursive calls. With `[memoize]`, each unique argument is computed only once — exponential time becomes linear.

How the three methods work together

`AstFunctionAnnotation` has several override points. `[memoize]` uses three of them in sequence:

1. ``**apply**()`` — runs when the annotation is first attached to a function, *before* type inference. We reject functions that cannot be memoized: generics (types are unknown), void returns (nothing to cache), and zero-argument functions (nothing to hash).
2. ``**patch**()`` — runs *after* type inference succeeds. All types are resolved, so we can build the cache table type and the wrapper function. Setting `astChanged = true` tells the compiler to restart inference so the new functions get type-checked. The "already processed" guard (`find_arg(args, "patched") is tBool`) ensures we don't regenerate on the second pass.
3. ``**transform**()`` — runs on *every call site* of the annotated function during inference. On the second pass (after `patch()` generated the wrapper), it replaces each call with a call to the memoized wrapper. On the first pass, it returns `default` to leave the call unchanged.

What patch() generates

For a function `fib(n : int) : int`, `patch()` produces three things:

A private copy of the original function (without `[memoize]`) so the wrapper can call it without triggering `transform()` again:

```
def private `memoize`original`fib(n : int) : int {
    if (n <= 1) { return n; }
    return fib(n - 1) + fib(n - 2)
}
```

A private global cache variable:

```
var private `memoize`cache`fib : table<uint64; int>
```

A private wrapper function that checks the cache, calls the original on miss, and stores the result via `insert_clone`:

```
def private `memoize`fib(n : int) : int {
    let key = hash(n)
    if (key_exists(`memoize`cache`fib, key)) {
        unsafe { return `memoize`cache`fib[key]; }
    }
    let result = `memoize`original`fib(n)
    `memoize`cache`fib |> insert_clone(key, result)
    return result
}
```

Then `transform()` redirects every call to `fib(...)` — including recursive calls inside `fib` itself — to \`memoize\`fib(...).

Module file: advanced_function_macro_mod.das

apply() — pre-inference validation

```
[function_macro(name="memoize")]
class MemoizeMacro : AstFunctionAnnotation {

    def override apply(var func : FunctionPtr; var group : ModuleGroup;
                       args : AnnotationArgumentList; var errors : das_string) : bool {
        if (func.isGeneric) {
            errors := "cannot memoize a generic function - all argument types must be␣
↪specified"
            return false
        }
        if (func.result.isVoid) {
            errors := "cannot memoize a void function - there is nothing to cache"
            return false
        }
        if (length(func.arguments) == 0) {
            errors := "cannot memoize a function with no arguments - there is nothing to␣
↪hash"
            return false
```

(continues on next page)

(continued from previous page)

```
        }
        return true
    }
```

`apply()` rejects functions at parse time — before the compiler has resolved types. The checks use pre-inference properties that are already available: `isGeneric`, `result.isVoid`, and `length(arguments)`.

patch() — code generation after inference

The "already processed" guard

```
def override patch(var fn : FunctionPtr; var group : ModuleGroup;
                   args, progArgs : AnnotationArgumentList;
                   var errors : das_string; var astChanged : bool&) : bool {

    // Guard: already processed?
    if (find_arg(args, "patched") is tBool) {
        return true
    }
```

Because `patch()` sets `astChanged = true`, inference restarts and `patch()` is called again. Without this guard, the macro would generate duplicate functions and hit an infinite loop.

Mark as processed and trigger restart

```
// Mark as processed and trigger inference restart
for (ann in fn.annotations) {
    if (ann.annotation.name == "memoize") {
        astChanged = true
        ann.arguments |> add_annotation_argument("patched", true)
    }
}
```

`add_annotation_argument` stores data in the annotation that persists across inference passes. We also store the wrapper function name later, so `transform()` can read it.

Step 1 — clone the original function

```
var inscope originalCopy <- clone_function(fn)
originalCopy.name := originalCopyName
originalCopy.flags |= FunctionFlags.generated | FunctionFlags.privateFunction
// Remove [memoize] from the clone to prevent infinite transform loop
let memoizeIdx = find_index_if(each(originalCopy.annotations)) $(ann) {
    return ann.annotation.name == "memoize"
}
if (memoizeIdx >= 0) {
    originalCopy.annotations |> erase(memoizeIdx)
```

(continues on next page)

(continued from previous page)

```
}
compiling_module() |> add_function(originalCopy)
```

The wrapper needs to call the *real* implementation. But `transform()` redirects *all* calls to the annotated function — including calls inside the wrapper. The solution is to clone the original, strip the `[memoize]` annotation from the clone, and have the wrapper call the unannotated copy.

Step 2 — create the cache variable

```
var inscope retType <- clone_type(fn.result)
retType.flags &= ~TypeDeclFlags.constant
retType.flags &= ~TypeDeclFlags.ref
var inscope wrapperRetType <- clone_type(retType)

var inscope keyType <- new TypeDecl(baseType = Type.tUInt64, at = fn.at)
var inscope cacheType <- new TypeDecl(baseType = Type.tTable, at = fn.at)
move(cacheType.firstType) <| keyType
move(cacheType.secondType) <| retType
add_global_var(compiling_module(), cacheName, clone_type(cacheType), fn.at, true)
```

The table type `table<uint64; RetType>` is built manually because `$t()` splicing doesn't work inside `typeinfo ast_typedecl` for table value types. `clone_type(cacheType)` is required because `add_global_var` takes ownership of the `TypeDeclPtr` without cloning it — if you pass an `inscope` variable directly, it gets deleted at scope exit and the compiler crashes on the next inference pass.

Step 4 — hash key computation

```
var inscope hashExprs : array<ExpressionPtr>
for (arg in fn.arguments) {
    hashExprs |> emplace_new <| qmacro(hash($i(arg.name)))
}

// Combine hashes with XOR
var inscope keyExpr <- hashExprs[0]
for (i in range(1, length(hashExprs))) {
    if (true) {
        var inscope xorExpr <- qmacro($e(keyExpr) ^ $e(hashExprs[i]))
        unsafe { keyExpr <- xorExpr; }
    }
}
```

For multiple arguments, the cache key is `hash(a) ^ hash(b) ^ ...`. In daslang, every type has a `hash()` function, so this works for strings, floats, structs, etc. The `if (true)` wrapper is a workaround for `var inscope` not being allowed directly in loop bodies.

Step 6 — assemble the wrapper body

```
var inscope bodyExprs : array<ExpressionPtr>
bodyExprs |> emplace_new <| qmacro_expr(${ let key = $e(keyExpr); })
bodyExprs |> emplace_new <| qmacro_expr(${ if (key_exists($i(cacheName), key)) { unsafe
↪{ return $i(cacheName)[key]; } } })
bodyExprs |> emplace_new <| qmacro_expr(${ let result = $c(originalCopyName)(
↪$a(callArgs)); })
bodyExprs |> emplace_new <| qmacro_expr(${ $i(cacheName) |> insert_clone(key, result); })
bodyExprs |> emplace_new <| qmacro_expr(${ return result; })
```

Each `qmacro_expr` generates one statement. The splicing operators:

- `$e(expr)` — splice an expression AST node
- `$i(name)` — splice a name as an identifier (`ExprVar`)
- `$c(name)` — splice a name into a call expression (`ExprCall`)
- `$a(array)` — splice an array of expressions as arguments
- `$t(type)` — splice a type declaration

Step 7–8 — create and add the wrapper function

```
var inscope wrapperFn <- qmacro_function(wrapperName) $($a(wrapperArgs)) :
↪$t(wrapperRetType) {
    $b(bodyExprs)
}
wrapperFn.flags |= FunctionFlags.generated | FunctionFlags.privateFunction
wrapperFn.body |> force_at(fn.body.at)
compiling_module() |> add_function(wrapperFn)

// Store the wrapper name for transform() to read
for (ann in fn.annotations) {
    if (ann.annotation.name == "memoize") {
        ann.arguments |> add_annotation_argument("wrapper", wrapperName)
    }
}
```

`qmacro_function` creates a new `FunctionPtr` with `$b(bodyExprs)` splicing the body statements. `force_at` adjusts source locations so error messages point to the original function. The wrapper name is stored in the annotation arguments — `transform()` reads it on the next pass.

transform() — call-site redirection

```
def override transform(var call : smart_ptr<ExprCallFunc>;
                       var errors : das_string) : ExpressionPtr {
    for (ann in call.func.annotations) {
        if (ann.annotation.name == "memoize") {
            let wrapperArg = find_arg(ann.arguments, "wrapper")
            if (wrapperArg is tString) {
```

(continues on next page)

(continued from previous page)

```
                let wrapperName = wrapperArg as tString
                var inscope newCall <- clone_expression(call)
                (newCall as ExprCall).name := wrapperName
                return <- newCall
            }
        }
    }
    return <- default<ExpressionPtr>
}
```

`transform()` is called for every call to the annotated function. It reads the wrapper name from the annotation, clones the call expression, changes the function name to the wrapper, and returns the replacement. When it returns a non-default value, the compiler automatically reports `astChanged` — no manual flag needed.

On the first inference pass (before `patch()` runs), the `"wrapper"` argument doesn't exist yet, so `transform()` returns `default` and the call goes through unchanged.

Usage file: `04_advanced_function_macro.das`

```
options gen2
require advanced_function_macro_mod

[memoize]
def fib(n : int) : int {
    if (n <= 1) { return n; }
    return fib(n - 1) + fib(n - 2)
}

[memoize]
def slow_add(a, b : int) : int {
    return a + b
}

[memoize]
def greet(name : string) : string {
    return "hello, {name}!"
}

[export]
def main() {
    print("fib(10) = {fib(10)}\n")
    print("fib(20) = {fib(20)}\n")
    print("fib(30) = {fib(30)}\n")
    print("slow_add(3, 4) = {slow_add(3, 4)}\n")
    print("slow_add(3, 4) = {slow_add(3, 4)}\n")  // cached
    print("{greet("daslang")}\n")
    print("{greet("daslang")}\n")  // cached
}
```

Output:

```
fib(10) = 55
fib(20) = 6765
fib(30) = 832040
slow_add(3, 4) = 7
slow_add(3, 4) = 7
hello, daslang!
hello, daslang!
```

Three functions are memoized: `fib` (recursive, single int argument), `slow_add` (multi-argument, hash XOR), and `greet` (string result, cloneable via `insert_clone`). The second calls to `slow_add` and `greet` hit the cache.

Compile-time error examples

The `apply()` method rejects invalid uses at compile time:

```
// ERROR: cannot memoize a void function - there is nothing to cache
// [memoize]
// def fire(x : int) { print("fire {x}\n"); }

// ERROR: cannot memoize a function with no arguments - there is nothing to hash
// [memoize]
// def get_zero() : int { return 0; }
```

Key techniques summary

Technique	What it does
`apply()`	Pre-inference validation: `isGeneric`, `isVoid`, argument count
`patch()`	Post-inference code generation with `astChanged` restart
`transform()`	Call-site redirection: clone expression, rename function
"Already processed" guard	`find_arg(args, "patched") is tBool` prevents double generation
`add_annotation_argument`	Stores data across inference passes
`clone_function`	Deep-clones a function with all annotations
`add_global_var`	Creates module-level variables at compile time (pass `clone_type`, not `inscope`)
`qmacro_function`	Reification: builds a function from spliced arguments, body, and return type
`qmacro_expr`	Reification: builds individual statements
`clone_expression`	Deep-clones an `ExpressionPtr`
`find_unique_function`	Checks whether a function already exists in a module
`insert_clone`	Table insertion that clones the value
`hash()` + XOR	Multi-argument cache key via `hash(a) ^ hash(b)`

See also:

Full source: `advanced_function_macro_mod.das`, `04_advanced_function_macro.das`

Previous tutorial: tutorial_macro_function_macro

Next tutorial: tutorial_macro_tag_function_macro

Language reference: *Macros* — full macro system documentation

5.4.5 Macro Tutorial 5: Tag Function Macros

In tutorials 3 and 4, `[function_macro]` required the macro class to live in a separate module compiled *before* the usage file. The annotation needs the class to exist at parse time, so two files are unavoidable.

daslang offers a lighter pattern — `[tag_function]` + `[tag_function_macro]` — that lets both the tagged function and its macro class live in the **same module**.

This tutorial builds a `once()` macro that executes a block only on the first call. Each call site gets its own auto-generated global boolean flag:

```
for (i in range(5)) {
    once() {
        print("runs exactly once\n")
    }
    print("  iteration {i}\n")
}
```

Output:

```
runs exactly once
  iteration 0
  iteration 1
  iteration 2
  iteration 3
  iteration 4
```

Why tag functions?

`[function_macro(name="X")]` expects the class `X` to already exist when the annotated function definition is parsed. This means the macro class and the tagged function **cannot** appear in the same compilation unit — you always need a two-file setup.

`[tag_function(tag_name)]` takes a different approach:

1. At parse time, the function is marked with a string tag. No class lookup happens yet.
2. During module setup, `[tag_function_macro(tag="tag_name")]` scans the module for all functions carrying the matching tag and programmatically attaches the macro class as their annotation.

Because the attachment happens *after* both the function and the class are compiled, a single module can contain everything.

This pattern is used by many standard library modules:

Module	Tags
`daslib/unroll`	Compile-time loop unrolling
`daslib/defer`	Go-style `defer` (move code to finally)
`daslib/assert_once`	Fire assertion only on first failure
`daslib/static_let`	Promote locals to hidden globals
`daslib/safe_addr`	Safe address-of operations
`daslib/jobque_boost`	`parallel_for`, `new_job`

In all these modules, the public function and the macro class coexist in one file — no extra "mod" module is needed.

Note: Our tutorial still uses two files because the *usage* file `require`-s the module, which is the normal deployment pattern. The key difference from `[function_macro]` is that the module itself is self-contained — you never need a *third* helper file just to define the macro class.

The module: `tag_function_macro_mod.das`

The module has two parts: the tagged function and the macro class.

Part 1 — The tagged function

```
[tag_function(once_tag)]
def public once(blk : block) {
    invoke(blk)
}
```

`[tag_function(once_tag)]` does two things:

- Records the string `"once_tag"` as a tag on this function.
- Does **not** attach any macro class — that happens later.

The function body (`invoke(blk)`) is a fallback: it runs only if the macro fails to transform the call for some reason. In normal operation, every call to `once()` is rewritten by `transform()` and the original body is never executed.

Part 2 — The macro class

```
[tag_function_macro(tag="once_tag")]
class OnceMacro : AstFunctionAnnotation {
    def override transform(var call : smart_ptr<ExprCallFunc>;
                           var errors : das_string) : ExpressionPtr {
        // ... rewrite every call to once()
    }
}
```

`[tag_function_macro(tag="once_tag")]` tells the compiler:

> *During module setup, find every function tagged with* `once_tag` *and attach this class as its function annotation.*

After setup, every call to `once()` triggers the `transform()` method exactly as if we had used `[function_macro]`.

Inside `transform()`

The method receives the call expression and returns a replacement AST. It proceeds in four steps.

Step 1 — Generate a unique flag name

```
let flag_name = make_unique_private_name("__once_flag", call.at)
```

`make_unique_private_name` combines the prefix with the call-site line and column numbers, producing names like `__once_flag_12_5`. Every call site gets its own name, so multiple `once()` calls in the same function are completely independent.

Step 2 — Create the global flag

```
if (!compiling_module() |> add_global_private_var(flag_name, call.at) <| quote(false)) {
    errors := "can't add global variable {flag_name}"
    return <- default<ExpressionPtr>
}
```

`add_global_private_var` inserts a private `bool` variable (initialized to `false` via `quote(false)`) into the module being compiled. The variable is private, so it never leaks into the public API.

If the variable already exists (e.g., the compiler re-runs inference), the function returns `false` and we report an error.

Step 3 — Extract the block body

```
var inscope block_clone <- clone_expression(call.arguments[0])
var inscope blk <- move_unquote_block(block_clone)
var inscope stmts : array<ExpressionPtr>
for (s in blk.list) {
    stmts |> emplace_new <| clone_expression(s)
}
```

When the user writes `once() { ... }`, the first argument is an `ExprMakeBlock` wrapping an `ExprBlock`. We:

1. Clone the argument expression (never modify the original AST).
2. Unwrap the `ExprMakeBlock` → `ExprBlock` via `move_unquote_block`.
3. Copy its statement list into a flat `array<ExpressionPtr>`.

The statements array is needed because the `$b()` splice operator expects `array<ExpressionPtr>`, not an `ExprBlock` directly.

Step 4 — Build the replacement

```
var inscope replacement <- qmacro_block() {
    if (!$i(flag_name)) {
        $i(flag_name) = true
        $b(stmts)
    }
}
replacement |> force_at(call.at)
return <- replacement
```

`qmacro_block` builds an `ExprBlock` using the reification mini-language:

- `$i(flag_name)` splices the string as an identifier reference.
- `$b(stmts)` splices the statement array into the `if` body.

`force_at` stamps every node in the replacement with the original call-site location so error messages point to the right place.

The final expansion of:

```
once() {
    print("hello\n")
}
```

is:

```
if (!__once_flag_12_5) {
    __once_flag_12_5 = true
    print("hello\n")
}
```

The usage file

```
options gen2
require tag_function_macro_mod

def test_loop() {
    for (i in range(3)) {
        once() {
            print("initialized (runs once)\n")
        }
        print("  iteration {i}\n")
    }
}

def test_multiple() {
    for (i in range(2)) {
        once() {
            print("first once (runs once)\n")
        }
        once() {
```

(continues on next page)

(continued from previous page)

```
            print("second once (runs once)\n")
        }
        print("  pass {i}\n")
    }
}

def greet() {
    once() {
        print("welcome! (runs once)\n")
    }
    print("  greet called\n")
}

[export]
def main() {
    print("--- test_loop ---\n")
    test_loop()
    print("\n--- test_multiple ---\n")
    test_multiple()
    print("\n--- test_greet ---\n")
    greet()
    greet()
    greet()
}
```

`test_loop` — each iteration checks the flag; only the first one fires. `test_multiple` — two `once()` calls have *different* flags (different line numbers), so both fire once. `greet` — the flag is global, so three separate calls still fire only once.

Full output:

```
--- test_loop ---
initialized (runs once)
  iteration 0
  iteration 1
  iteration 2

--- test_multiple ---
first once (runs once)
second once (runs once)
  pass 0
  pass 1

--- test_greet ---
welcome! (runs once)
  greet called
  greet called
  greet called
```

Key takeaways

Concept	What it does
`[tag_function(tag)]`	Marks a function with a string tag — no class lookup at parse time
`[tag_function_macro(tag="..")]`	During module setup, attaches the macro class to all functions with matching tag
`transform()`	Called at every call site; returns a replacement `ExpressionPtr`
`make_unique_private_name`	Generates `prefix_LINE_COLUMN` names unique to each call site
`add_global_private_var`	Creates a private mutable module-level variable at compile time
`clone_expression`	Deep-clones an AST node (never mutate the original)
`move_unquote_block`	Unwraps `ExprMakeBlock` → `ExprBlock`
`qmacro_block`	Reification: builds a block from spliced identifiers and statements
`$i(name)`	Splice a string as an identifier
`$b(stmts)`	Splice `array<ExpressionPtr>` as a statement list
`force_at`	Stamps source location on all nodes

See also:

Full source: `tag_function_macro_mod.das`, `05_tag_function_macro.das`

Previous tutorial: tutorial_macro_advanced_function_macro

Next tutorial: tutorial_macro_structure_macro

Standard library examples: `daslib/assert_once.das` (closest to our `once()`), `daslib/unroll.das`, `daslib/defer.das`, `daslib/static_let.das`

Language reference: *Macros* — full macro system documentation

5.4.6 Macro Tutorial 6: Structure Macros

Tutorials 3–5 transformed *function calls*. Structure macros operate on **struct and class definitions** instead.

`[structure_macro(name="X")]` registers a class that extends `AstStructureAnnotation`. When a struct is annotated with `[X]`, the compiler calls the macro's methods at three stages of the compilation pipeline:

Method	When it runs
`apply()`	During parsing, **before** inference. Can add fields, validate annotation arguments, and generate new functions.
`patch()`	**After** inference. Types are resolved, so type-aware checks are possible. Can set `astChanged` to restart inference.
`finish()`	After all inference and optimization. Read-only — useful for diagnostics.

This tutorial builds a `[serializable]` annotation that:

1. Adds a `_version` field and generates a **stub** `describe_StructName()` function — header only (`apply`).
2. Fills in the describe function body with per-field printing, **skipping** non-serializable types like function pointers and lambdas — only possible after inference (`patch`).
3. Prints a compile-time summary of each struct (`finish`).

```
[serializable(version=2)]
struct Player {
    name : string
```

(continues on next page)

(continued from previous page)

```
    health : int
    on_hit : function<(damage:int):void>   // skipped by describe
}
```

After compilation the struct gains a `_version : int = 2` field and a generated `describe_Player` function that prints `name` and `health` but skips `on_hit` (a function pointer).

Why three methods?

Each method runs at a different point in the pipeline, which determines what it can and cannot do:

- **apply()** — The struct definition is parsed but types are not resolved. You can add fields and generate functions, but you cannot inspect `fld._type.baseType` because types are still `autoinfer`. Any generated functions **must** be created here so they exist when inference runs — but their bodies can be stubs that `patch()` fills in later.
- **patch()** — Runs after inference, so all types are resolved. This is the place for type-aware code generation or validation. After modifying a function body, set `astChanged = true` to restart inference. Use a guard to avoid infinite loops (e.g., check whether the body length changed since the stub was created).
- **finish()** — Everything is final: types are resolved, code is optimized. No modifications allowed. Use it for diagnostics, compile-time reporting, or AOT-related output.

The module: `structure_macro_mod.das`

Registration

```
[structure_macro(name="serializable")]
class SerializableMacro : AstStructureAnnotation {
    ...
}
```

`[structure_macro(name="serializable")]` tells the compiler:

> *When a struct or class is annotated with* `[serializable]`*, call this class's methods during compilation.*

Like `[function_macro]`, the macro class must be compiled before any module that uses the annotation — hence the two-file setup.

Inside `apply()`

The method receives the struct definition, annotation arguments, and an error string. It runs during parsing, before inference.

Step 1 — Validate arguments

```
var version = 1
for (arg in args) {
    if (arg.name == "version") {
        let val = get_annotation_argument_value(arg)
        if (val is tInt) {
            version = val as tInt
        } else {
            errors := "[serializable] 'version' argument must be an integer"
            return false
        }
    } else {
        errors := "[serializable] unknown argument - only 'version' is supported"
        return false
    }
}
```

We iterate the `AnnotationArgumentList` and accept only `version` (integer). Returning `false` aborts compilation with the error message stored in `errors`.

Step 2 — Add a field

```
st |> add_structure_field("_version",
    clone_type(qmacro_type(type<int>)),
    qmacro($v(version)))
```

`add_structure_field` appends a new field to the struct's field list. It **moves** both the `TypeDeclPtr` and `ExpressionPtr` arguments, so they must be either temporaries or clones.

> **Warning:** Never pass a `var inscope` variable directly to `add_structure_field` — it will be moved *and* destroyed at scope exit, causing a double-free crash. Always pass `clone_type(...)` or an inline temporary.

`qmacro_type(type<int>)` creates a `TypeDeclPtr` for `int`. `qmacro($v(version))` creates an integer constant expression.

Step 3 — Generate a stub describe function

```
let funcName = "describe_{st.name}"
var inscope bodyExprs : array<ExpressionPtr>

bodyExprs |> emplace_new <| qmacro(print($v("{st.name} (version ")))
bodyExprs |> emplace_new <| qmacro(print("{obj._version}"))
bodyExprs |> emplace_new <| qmacro(print($v("):\n")))

var inscope fn <- qmacro_function(funcName) $(obj : $t(st)) {
    $b(bodyExprs)
}
fn.flags |= FunctionFlags.generated
```

(continues on next page)

(continued from previous page)

```
fn.body |> force_at(st.at)
add_function(st._module, fn)
```

The stub function prints only the header line — the struct name and version. **Per-field printing is deferred to `patch()` where types are known.**

The function must exist before inference so callers (like `main`) can reference `describe_Color()` or `describe_Player()`. Its body will be extended in `patch()` after types resolve.

Two kinds of content appear in `qmacro`:

- **Compile-time constants** — field names, struct name. Use `$v("string")` to splice a constant string into the generated code. String interpolation like `"{st.name}"` resolves at *macro execution time* and becomes a literal.
- **Runtime values** — `obj._version`, `obj.$f(fld.name)`. Inside `qmacro`, `obj` refers to the generated function's parameter. `$f(fld.name)` splices a string as a field-access name. To convert any value to a string for `print`, use string interpolation: `print("{obj.$f(fld.name)}")`.

`qmacro_function(funcName) $(obj : $t(st)) { $b(bodyExprs) }` builds a complete function:

- `funcName` — a string for the function name
- `$t(st)` — splices the struct type as the parameter type
- `$b(bodyExprs)` — splices the statement array into the body

`add_function(st._module, fn)` registers the function in the struct's module so callers can find it.

Inside `patch()`

This is the heart of the tutorial. In `apply()` we created a stub function — now we fill it in, using inferred type information to skip non-serializable fields.

Step 1 — Guard against re-patching

```
if (find_arg(args, "patched") is tBool) {
    return true
}
```

Setting `astChanged` causes the compiler to re-run inference, which calls `patch()` again. Without a guard, this would loop forever. We add a `"patched"` annotation argument (in Step 5) and check for it here — if present, the work is already done.

Step 2 — Find the stub function

```
let funcName = "describe_{st.name}"
var inscope fn <- st._module |> find_unique_function(funcName)
```

`find_unique_function` (from `daslib/ast_boost`) searches a module for a function by name. It returns a `smart_ptr<Function>` pointing to the same object in the module — modifications through this pointer affect the actual function.

Step 3 — Get the body as ExprBlock

```
unsafe {
    var blk = reinterpret<ExprBlock?> fn.body
```

The function body is an `ExpressionPtr` internally — we know it is an `ExprBlock` because we built it that way in `apply()`. `reinterpret<ExprBlock?>` (requires `unsafe`) gives us a typed pointer so we can access the `list` array of statements.

Step 4 — Append field-printing statements

```
    for (fld in st.fields) {
        if (fld.name == "_version") {
            continue
        }
        if (fld._type.baseType == Type.tLambda ||
            fld._type.baseType == Type.tFunction) {
            continue
        }
        blk.list |> emplace_new <| qmacro(print($v("  {fld.name} = ")))
        blk.list |> emplace_new <| qmacro(print("{obj.$f(fld.name)}"))
        blk.list |> emplace_new <| qmacro(print($v("\n")))
    }
}
```

Now `fld._type.baseType` is resolved — this was `autoinfer` in `apply()` but is the real type here. We skip fields whose type is lambda or function pointer (blocks cannot appear as struct fields, so no `tBlock` check is needed). The `obj` reference works because the generated expressions will be re-inferred inside the function where `obj` is a parameter.

Step 5 — Mark as patched and trigger re-inference

```
for (ann in st.annotations) {
    if (ann.annotation.name == "serializable") {
        ann.arguments |> add_annotation_argument("patched", true)
    }
}
astChanged = true
```

We add a `"patched"` boolean argument to our own annotation — this is the marker that Step 1 checks on the next pass. Then `astChanged = true` tells the compiler to re-run inference on the modified function body. On the next pass, `find_arg(args, "patched")` returns `tBool` and `patch()` returns immediately.

Inside `finish()`

```
def override finish(var st : StructurePtr; var group : ModuleGroup;
                    args : AnnotationArgumentList; var errors : das_string) : bool {
    var serializable = 0
    var skipped = 0
    for (fld in st.fields) {
        if (fld.name == "_version") {
            continue
        }
        if (fld._type.baseType == Type.tLambda ||
            fld._type.baseType == Type.tFunction) {
            skipped++
        } else {
            serializable++
        }
    }
    print("[serializable] {st.name}: {serializable} serializable field(s)")
    if (skipped > 0) {
        print(", {skipped} skipped")
    }
    // ... print version
    return true
}
```

`finish()` runs after all inference and optimization. The struct is in its final form — we count serializable and skipped fields and print a compile-time diagnostic.

`find_arg(args, "version")` returns an `RttiValue` variant, checked with `is tInt` / `as tInt`.

The usage file

```
options gen2
require structure_macro_mod

[serializable]
struct Color {
    r : float
    g : float
    b : float
}

[serializable(version=2)]
struct Player {
    name : string
    health : int
    score : float
    on_hit : function<(damage:int):void>
}

[export]
def main() {
```

(continues on next page)

(continued from previous page)

```
    var c = Color(r = 0.2, g = 0.7, b = 1.0)
    describe_Color(c)

    var p = Player(name = "Alice", health = 100, score = 42.5)
    describe_Player(p)

    print("Color version: {c._version}\n")
    print("Player version: {p._version}\n")
}
```

`Color` uses the default version (1) — all fields are plain types. `Player` specifies `version=2` and has an `on_hit` function pointer field. The generated `describe_Player` prints `name`, `health`, and `score` but **skips** `on_hit` because `patch()` detected it as `Type.tFunction`.

Compile-time output (from `finish`):

```
[serializable] Color: 3 serializable field(s), version 1
[serializable] Player: 3 serializable field(s), 1 skipped, version 2
```

Runtime output:

```
--- describe_Color ---
Color (version 1):
  r = 0.2
  g = 0.7
  b = 1

--- describe_Player ---
Player (version 2):
  name = Alice
  health = 100
  score = 42.5

--- version info ---
Color version: 1
Player version: 2
```

Compilation pipeline summary

The full sequence for a `[serializable]` struct:

```
parse struct definition
  ↓
apply() → add _version field, generate stub describe_X()
  ↓
infer types (stub function is inferred with header-only body)
  ↓
patch() → find stub, append field prints (skip bad types), mark "patched", astChanged
  ↓
re-infer (modified body now has field-specific print statements)
  ↓
patch() → "patched" arg found → return immediately
```

(continues on next page)

(continued from previous page)

```
  ↓
optimize
  ↓
finish() → compile-time diagnostic (serializable vs skipped)
  ↓
simulate → run
```

Key takeaways

Concept	What it does
`[structure_macro(name="X")]`	Registers a class as a structure annotation
`AstStructureAnnotation`	Base class with `apply`, `patch`, `finish` methods
`apply()`	Pre-inference: add fields, generate stub functions
`patch()`	Post-inference: fill in bodies using type info, set `astChanged`
`finish()`	Final: read-only diagnostics and reporting
`astChanged`	Set `true` in `patch` to restart inference after changes
`add_annotation_argument`	Marks the annotation as processed to prevent infinite re-patching
`find_unique_function`	Locates a function by name in a module (`ast_boost`)
`reinterpret<ExprBlock?>`	Casts `fn.body` to `ExprBlock?` to access the statement list
`add_structure_field`	Appends a field to a struct; moves both type and init expression
`clone_type`	Deep-clones a `TypeDeclPtr`; required before move operations
`qmacro_function`	Builds a complete function from reification splices
`$v(value)`	Splice a compile-time value as a constant expression
`$f(name)`	Splice a string as a field-access name
`$t(type)`	Splice a `TypeDeclPtr` into parameter/return types
`$b(stmts)`	Splice `array<ExpressionPtr>` as a statement list
`find_arg`	Look up annotation argument values by name
`force_at`	Stamps source location on all generated AST nodes

See also:

Full source: `structure_macro_mod.das`, `06_structure_macro.das`

Previous tutorial: tutorial_macro_tag_function_macro

Next tutorial: tutorial_macro_block_macro

Standard library examples: `daslib/interfaces.das` (`apply` + `finish`), `daslib/decs_boost.das` (`apply` with field iteration)

Language reference: *Macros* — full macro system documentation

5.4.7 Macro Tutorial 7: Block Macros

Tutorials 3–6 transformed *functions* and *structs*. Block macros operate on **block closures** instead — the `$ { ... }` expressions that are passed to functions as arguments.

`[block_macro(name="X")]` registers a class that extends `AstBlockAnnotation`. When a block is annotated with `[X]`, the compiler calls the macro's methods at two stages of the compilation pipeline:

Method	When it runs
`apply()`	During parsing, **before** inference. Can modify the block's statement list and finally list, validate annotation arguments, and inject code.
`finish()`	After all inference and optimization. Read-only — argument types are fully resolved, useful for diagnostics.

This tutorial builds a `[traced(tag="X")]` annotation that:

1. Prepends an enter-message and appends an exit-message (via `finalList`) to the block body — the exit message runs even on early return (`apply`).
2. Prints a compile-time summary of each block's typed arguments and statement count (`finish`).

```
run_block() $ [traced(tag="setup")] {
    print("  initializing\n")
}
```

After compilation the block body gains `print(">> setup\n")` at the start and `print("<< setup\n")` in its finally section, so every invocation prints entry and exit markers around the user code.

Block annotation syntax

Block annotations are placed between the `$` sigil and the parameter list (or body, for parameterless blocks):

```
// Parameterless block
$ [annotation(args)] { body }

// Block with parameters
$ [annotation(args)] (params) { body }
```

Multiple annotations can be comma-separated inside the brackets, just like function annotations:

```
$ [traced(tag="x"), REQUIRE(hp)] (v : int) { ... }
```

Why only two methods?

Structure macros have three methods (`apply`, `patch`, `finish`) because structs often need type-aware code generation in `patch()` after inference resolves field types. Block macros skip `patch()` entirely:

- **apply()** — The block is parsed but types may not be resolved. You can modify `blk.list` (the statement list) and `blk.finalList` (the finally section). Validation and code injection happen here.
- **finish()** — Everything is final: argument types are resolved, code is optimized. No modifications allowed. Use it for diagnostics, compile-time reporting, or verifying block structure.

Blocks are self-contained expressions — simpler than functions or structs — so two methods are sufficient. If you need post-inference code generation, use a function macro or structure macro instead.

The module: `block_macro_mod.das`

Registration

```
[block_macro(name="traced")]
class TracedBlockMacro : AstBlockAnnotation {
    ...
}
```

`[block_macro(name="traced")]` tells the compiler:

> *When a block is annotated with* `[traced]`*, call this class's methods during compilation.*

Block annotations are registered as function annotations under the hood — `add_new_block_annotation` internally calls `add_function_annotation`. This is why block annotations share the `AnnotationArgumentList` parameter type with function macros.

Inside `apply()`

The method receives the block expression, annotation arguments, and an error string. It runs during parsing, before inference.

Step 1 — Validate arguments

```
let labelArg = find_arg(args, "tag")
if (!(labelArg is tString)) {
    errors := "[traced] requires a 'tag' string argument"
    return false
}
let lbl = labelArg as tString
```

We use `find_arg` (from `daslib/ast_boost`) to look up the `tag` argument by name. It returns an `RttiValue` variant — we check `is tString` and cast with `as tString`. Returning `false` aborts compilation with the error message.

Step 2 — Prepend enter-print

```
var inscope enterExpr <- qmacro(print($v(">> {lbl}\n")))
blk.list |> emplace(enterExpr, 0)
```

`blk.list` is the block's statement array. `emplace(vec, val, 0)` inserts at position 0, pushing existing statements down. This places the enter-print **before** any user code.

`$v(">> {lbl}\n")` splices the compile-time string (with the tag value baked in) as a constant expression in the generated code.

Step 3 — Append exit-print to `finalList`

```
var inscope exitExpr <- qmacro(print($v("<< {lbl}\n")))
blk.finalList |> emplace(exitExpr)
```

`blk.finalList` is the block's **finally** section — statements here run when the block exits, regardless of how it exits. This guarantees the exit message prints even if the block has an early return.

Note: `finalList` on loop blocks (`blockFlags.inTheLoop`) only runs once at loop exit, not per iteration. For block closures passed to functions (like in this tutorial), `finalList` runs on every `invoke`, which is correct for enter/exit tracing.

Inside `finish()`

```
def override finish(var blk : smart_ptr<ExprBlock>; var group : ModuleGroup;
                    args, progArgs : AnnotationArgumentList;
                    var errors : das_string) : bool {
    let labelArg = find_arg(args, "tag")
    var lbl = "?"
    if (labelArg is tString) {
        lbl = labelArg as tString
    }

    // Count original statements (subtract injected enter-print)
    var numStmts = 0
    for (s in blk.list) {
        numStmts++
    }
    if (numStmts > 0) {
        numStmts -= 1
    }

    print("[traced] \"{lbl}\": {numStmts} statement(s)")

    // Iterate typed arguments - types are now resolved
    var numArgs = 0
    for (a in blk.arguments) {
        numArgs++
    }
    if (numArgs > 0) {
        print(", args = (")
        var first = true
        for (arg in blk.arguments) {
            if (!first) {
                print(", ")
            }
            print("{arg.name}:{describe(arg._type)}")
            first = false
        }
        print(")")
```

(continues on next page)

(continued from previous page)

```
    }

    print("\n")
    return true
}
```

`finish()` runs after all inference and optimization. The block is in its final form — argument types are fully resolved. We iterate `blk.arguments` (a managed vector of `VariablePtr`) and call `describe(arg._type)` to get human-readable type names like `int const`.

We count statements by iterating `blk.list` and subtract one to account for the enter-print injected by `apply()`.

The usage file

```
options gen2
require block_macro_mod

def run_block(blk : block) {
    invoke(blk)
}

def apply_to(x : int; blk : block<(v : int) : void>) {
    invoke(blk, x)
}

[export]
def main() {
    print("--- simple block ---\n")
    run_block() $ [traced(tag="setup")] {
        print("  initializing\n")
    }

    print("\n--- block with argument ---\n")
    apply_to(42) $ [traced(tag="process")] (v : int) {
        print("  received {v}\n")
    }
}
```

The first block is parameterless — it demonstrates the basic `[traced]` syntax. The second block takes an `int` parameter, showing that `finish()` can report its fully-typed arguments.

Both blocks are invoked by helper functions via `invoke(blk)` / `invoke(blk, x)`. Each invocation executes the modified block body (with the injected enter/exit prints).

Compile-time output (from `finish`):

```
[traced] "setup": 1 statement(s)
[traced] "process": 1 statement(s), args = (v:int const)
```

Runtime output:

```
--- simple block ---
>> setup
```

(continues on next page)

(continued from previous page)

```
  initializing
<< setup

--- block with argument ---
>> process
  received 42
<< process
```

Compilation pipeline summary

The full sequence for a `[traced]` block:

```
parse block closure
  ↓
apply() → validate tag, prepend enter-print, append exit-print to finalList
  ↓
infer types (block arguments and body are inferred)
  ↓
optimize
  ↓
finish() → compile-time diagnostic (statement count, typed args)
  ↓
simulate → invoke block at runtime (enter-print, user code, exit-print)
```

Unlike structure macros, there is no `patch` → `re-infer` cycle. All code injection happens in `apply()` before the first inference pass.

Key takeaways

Concept	What it does
`[block_macro(name="X")]`	Registers a class as a block annotation
`AstBlockAnnotation`	Base class with `apply` and `finish` methods (no `patch`)
`apply()`	Pre-inference: modify statement list, inject code, validate args
`finish()`	Final: read-only diagnostics and reporting with resolved types
`blk.list`	The block's statement array — modify to inject code
`blk.finalList`	The block's finally section — runs on exit, like a `finally` block
`blk.arguments`	The block's parameter list (`VariablePtr` elements)
`emplace(vec, val, 0)`	Insert at a specific position in a managed vector
`find_arg`	Look up annotation argument values by name
`describe(arg._type)`	Human-readable type name from a `TypeDeclPtr`
`$ [ann(args)] (params) { body }`	Block annotation syntax — annotation between `$` and parameters

See also:

Full source: `block_macro_mod.das`, `07_block_macro.das`

Previous tutorial: tutorial_macro_structure_macro

Next tutorial: tutorial_macro_variant_macro

Standard library examples: `daslib/decs_boost.das` (`REQUIRE` / `REQUIRE_NOT` — marker block annotations), `daslib/defer.das` (`finalList` manipulation), `daslib/heartbeat.das` (`emplace(list, expr, 0)` prepend pattern)

Language reference: *Macros* — full macro system documentation

5.4.8 Macro Tutorial 8: Variant Macros

Tutorials 3–7 transformed *functions*, *structs*, and *blocks*. Variant macros are different: they intercept the **``is``**, **``as``**, and **``?as``** operators during type inference and replace them with arbitrary expressions.

`[variant_macro(name="X")]` registers a class that extends `AstVariantMacro`. When the compiler encounters `expr is Name`, `expr as Name`, or `expr ?as Name`, it calls every registered variant macro's visitor method in order. The first one to return a non-default `ExpressionPtr` wins — the operator is replaced with the returned expression.

Three-tier resolution

The `is` / `as` / `?as` operators go through three resolution stages:

```
expr is/as/?as Name
  ↓
1. Variant macros  (visitExpr*Variant returns non-default)
  ↓
2. Generic operator functions  (def operator is/as Name)
  ↓
3. Built-in variant handling  (native variant types)
```

If a variant macro claims the expression, stages 2 and 3 are never reached. If all macros return `default<ExpressionPtr>`, the compiler tries generic `operator is`/`operator as` overloads, and finally falls through to built-in `variant` type dispatching.

`AstVariantMacro` methods

Method	Intercepts
`visitExprIsVariant(prog, mod, e)`	`expr is Name` — type check
`visitExprAsVariant(prog, mod, e)`	`expr as Name` — type cast
`visitExprSafeAsVariant(prog, mod, e)`	`expr ?as Name` — null-safe cast

Each method receives the full `ProgramPtr` and `Module?` context, plus the expression node. Return `default<ExpressionPtr>` to *decline* and let the next macro (or built-in logic) handle it. Return any other expression to *claim* the operator — the compiler replaces the original node with your expression.

Case study: `InterfaceAsIs`

This tutorial uses the `InterfaceAsIs` variant macro from `daslib/interfaces`. It makes the `is`, `as`, and `?as` operators work with `[interface]` types — no separate module file is needed because the macro lives in the standard library.

`daslib/interfaces` already provides two structure macros:

- `[interface]` — marks a class as an interface (function-only fields)
- `[implements(IFoo)]` — generates a proxy class and a `get`IFoo` getter method on the implementing struct

The `InterfaceAsIs` variant macro builds on top of these generated getters.

Type guard pattern

Every visitor method starts with a *type guard* — a series of checks that decide whether this macro should handle the expression:

```
def override visitExprIsVariant(prog : ProgramPtr; mod : Module?;
                                expr : smart_ptr<ExprIsVariant>) : ExpressionPtr {
    assume vtype = expr.value._type
    // 1. Value must be a pointer to a structure
    if (!(vtype.isPointer && vtype.firstType != null && vtype.firstType.isStructure)) {
        return <- default<ExpressionPtr>
    }
    // 2. Target name must refer to an [interface]-annotated struct
    let iname = string(expr.name)
    var tgt = prog |> find_unique_structure(iname)
    if (tgt == null || !is_interface_struct(tgt)) {
        return <- default<ExpressionPtr>
    }
    ...
}
```

If any check fails, the method returns `default<ExpressionPtr>` — declining to handle this expression and letting the next resolution stage take over.

`is` — compile-time check

Once the guard passes, `visitExprIsVariant` looks for a `get`IFoo` field on the source struct. If found, the struct implements the interface → return `true`. Otherwise → `false`:

```
let getter_field = "get`{iname}"
var st = vtype.firstType.structType
for (fld in st.fields) {
    if (string(fld.name) == getter_field) {
        return <- qmacro(true)
    }
}
return <- qmacro(false)
```

The result is a **compile-time constant** — no runtime cost at all. `w is IDrawable` becomes the literal `true` in the final program.

as — get interface proxy

visitExprAsVariant generates a call to the getter function:

```
let func_name = "{st.name}`get`{iname}"
return <- qmacro($c(func_name)(*$e(expr.value)))
```

$c(func_name) creates a function call by name (e.g. Widget`get`IDrawable). *$e(expr.value) dereferences the pointer to pass the struct by reference.

So w as IDrawable becomes Widget`get`IDrawable(*w), which returns an IDrawable? proxy.

?as — null-safe access

visitExprSafeAsVariant adds a null check before calling the getter:

```
var inscope val <- clone_expression(expr.value)
return <- qmacro($e(val) != null ? $c(func_name)(*$e(expr.value)) : null)
```

clone_expression is needed because the value expression appears twice — once in the null check and once in the getter call. The original expr.value is *moved* into the $e() splice, so a clone provides the second copy.

w ?as IDrawable becomes:

```
w != null ? Widget`get`IDrawable(*w) : null
```

The usage file

```
options gen2
require daslib/interfaces

[interface]
class IDrawable {
    def abstract draw(x, y : int) : void
}

[interface]
class IResizable {
    def abstract resize(w, h : int) : void
}

[implements(IDrawable), implements(IResizable)]
class Widget {
    def Widget() { pass }
    def IDrawable`draw(x, y : int) {
        print("Widget.draw at ({x},{y})\n")
    }
    def IResizable`resize(w, h : int) {
        print("Widget.resize to {w}x{h}\n")
    }
}
```

(continues on next page)

(continued from previous page)

```
[implements(IDrawable)]
class Label {
    text : string
    def Label(t : string) { text = t }
    def IDrawable`draw(x, y : int) {
        print("Label \"{text}\" at ({x},{y})\n")
    }
}
```

`Widget` implements both `IDrawable` and `IResizable`. `Label` implements only `IDrawable`. The `InterfaceAsIs` macro handles all three operators automatically:

```
[export]
def main() {
    var w = new Widget()
    var l = new Label("hello")

    // is - compile-time check
    print("w is IDrawable  = {w is IDrawable}\n")   // true
    print("l is IResizable = {l is IResizable}\n")   // false

    // as - get interface proxy
    var drawable = w as IDrawable
    drawable->draw(10, 20)

    // ?as - null-safe access
    var maybe_draw = l ?as IDrawable
    if (maybe_draw != null) {
        maybe_draw->draw(5, 5)
    }

    // null pointer - ?as returns null
    var nothing : Label?
    var safe = nothing ?as IDrawable
    print("null ?as IDrawable = {safe}\n")

    unsafe { delete w; delete l }
}
```

Runtime output:

```
w is IDrawable  = true
w is IResizable = true
l is IDrawable  = true
l is IResizable = false
Widget.draw at (10,20)
Widget.resize to 800x600
Label "hello" at (5,5)
null ?as IDrawable = null
```

Compilation pipeline

The full sequence for `w is IDrawable`:

```
parse expression (ExprIsVariant with value=w, name="IDrawable")
  ↓
infer types → value type is Widget?
  ↓
call visitExprIsVariant() on each registered variant macro
  ↓
InterfaceAsIs checks: Widget? → pointer to struct ✓
"IDrawable" is an [interface] ✓, Widget has get`IDrawable field ✓
  ↓
return qmacro(true) - replaces ExprIsVariant with ExprConstBool
  ↓
simulate → constant folded, zero runtime cost
```

Existing variant macros

The standard library ships several variant macros. Studying them is the best way to learn the pattern:

Macro	Purpose
`BetterRttiVisitor` (`daslib/ast_boost`)	Optimizes `is` on AST expression types to `__rtti` string comparison
`ClassAsIs` (`daslib/dynamic_cast_rtti`)	Dynamic casting for classes: `is` checks RTTI, `as`/`?as` cast pointers
`BetterJsonMacro` (`daslib/json_boost`)	`is`/`as` on `JsValue` to access JSON node types
`InterfaceAsIs` (`daslib/interfaces`)	`is`/`as`/`?as` for `[interface]` types (this tutorial)

Key takeaways

Concept	What it does
`[variant_macro(name="X")]`	Registers a class as a variant macro
`AstVariantMacro`	Base class with `visitExprIsVariant`, `visitExprAsVariant`, `visitExprSafeAsVariant`
`visitExprIsVariant`	Intercepts `expr is Name` — return replacement or default to decline
`visitExprAsVariant`	Intercepts `expr as Name`
`visitExprSafeAsVariant`	Intercepts `expr ?as Name`
`default<ExpressionPtr>`	Return value meaning "I don't handle this" — pass to next resolver
Type guard pattern	Check `expr.value._type` before claiming an expression
`find_unique_structure`	Look up a struct by name from the compiling program
`qmacro(true)`	Generate a compile-time constant — zero runtime cost
`$c(name)(args)`	Generate a function call by name in `qmacro`
`clone_expression`	Deep-copy an expression for safe double use in `qmacro`

See also:

Full source: `08_variant_macro.das`

Previous tutorial: tutorial_macro_block_macro

Next tutorial: tutorial_macro_for_loop_macro

Standard library: `daslib/interfaces.das` (`InterfaceAsIs` macro), `daslib/ast_boost.das` (`BetterRttiVisitor`), `daslib/dynamic_cast_rtti.das` (`ClassAsIs`), `daslib/json_boost.das` (`BetterJsonMacro`)

Language reference: *Macros* — full macro system documentation

5.4.9 Macro Tutorial 9: For-Loop Macros

Previous tutorials transformed calls, functions, structures, blocks, and variants. For-loop macros operate on **for-loop expressions** — they intercept `for (... in ...)` at compile time and can rewrite the entire loop before type inference finalises.

`[for_loop_macro(name="X")]` registers a class that extends `AstForLoopMacro`. The compiler calls the macro's single method after each for-loop's sources have been type-checked:

`visitExprFor(prog, mod, expr)`
: Called after visiting the for-loop body, during type inference. Source types are resolved. Return a replacement `ExpressionPtr` to transform the loop; return `default<ExpressionPtr>` to skip.

Motivation

daslang tables (`table<K;V>`) are not directly iterable. The standard idiom to iterate a table requires the verbose `keys()` and `values()` built-in functions:

```
for (k, v in keys(tab), values(tab)) {
    print("{k} => {v}\n")
}
```

This tutorial builds a for-loop macro that supports a much more natural tuple-destructuring syntax:

```
for ((k,v) in tab) {        // rewrites to keys/values automatically
    print("{k} => {v}\n")
}
```

The `(k,v)` syntax is **already supported by the parser** — it produces a backtick-joined iterator name (`k`v`) and sets a tuple-expansion flag. All the macro needs to do is detect a table source, split the name, and rewrite the loop.

Note: Macros cannot be used in the module that defines them. This tutorial has **two** source files: a *module* file containing the macro definition and a *usage* file that requires the module and exercises it.

The macro module

Prerequisites

```
require ast                // AST node types (ExprFor, ExprCall, etc.)
require daslib/ast_boost   // AstForLoopMacro base class, [for_loop_macro]
require strings            // find, slice - for splitting the iterator name
```

Step 1 — Registration

```
[for_loop_macro(name=table_kv)]
class TableKVForLoop : AstForLoopMacro {
    //! Transforms  for ((k,v) in tab)  into  for (k, v in keys(tab), values(tab)).
    def override visitExprFor(prog : ProgramPtr; mod : Module?; expr : smart_ptr<ExprFor>
↪) : ExpressionPtr {
```

`[for_loop_macro(name=table_kv)]` registers `TableKVForLoop` so the compiler calls `visitExprFor` for every for-loop in modules that `require` this one.

Step 2 — Detect a table source with tuple expansion

Each for-loop's `ExprFor` node has several parallel vectors:

sources
: Source expressions (the `in` parts).

iterators
: Iterator variable names.

iteratorsAt
: Source locations for each iterator.

iteratorsAka
: Alias names (from `aka`).

iteratorsTags
: Tag expressions.

iteratorsTupleExpansion
: `uint8` flag per iterator — nonzero if the parser saw `(k,v)` syntax.

iteratorVariables
: Resolved variables (populated by inference, cleared on rewrite).

The macro scans for a source where the tuple-expansion flag is set **and** the source type is a table:

```
        var tab_index = -1
        for (index, src in count(), expr.sources) {
            if (index < int(expr.iteratorsTupleExpansion |> length)) {
                if (int(expr.iteratorsTupleExpansion[index]) != 0 && src._type != null &&
↪ src._type.isGoodTableType) {
                    tab_index = index
                    break
                }
```

(continues on next page)

(continued from previous page)

```
            }
        }
```

Step 3 — Split the backtick-joined name

When the user writes `(k,v)`, the parser stores the iterator name as `"k`v"` (joined with a backtick). The macro splits this into `key_name` and `val_name`:

```
        let joined_name = string(expr.iterators[tab_index])
        let bt = find(joined_name, "`")
        if (bt < 0 || find(joined_name, "`", bt + 1) >= 0) {
            return <- default<ExpressionPtr>  // need exactly 2 parts
        }
        let key_name = slice(joined_name, 0, bt)
        let val_name = slice(joined_name, bt + 1)
```

Step 4 — Clone and rewrite

The transformation follows the same pattern as the standard library's `SoaForLoop` (in `daslib/soa.das`):

1. **Clone** the entire `ExprFor`.
2. **Erase** the table entry at `tab_index` from all parallel vectors.
3. **Add** two new sources — `keys(tab)` and `values(tab)` — with the split iterator names.
4. **Clear** `iteratorVariables` so inference rebuilds them on the next pass.

```
        // Clone the for expression
        var inscope new_for_e <- clone_expression(expr)
        var new_for = new_for_e as ExprFor
        // Erase the table entry from all parallel vectors
        let source_at = expr.sources[tab_index].at
        new_for.sources |> erase(tab_index)
        new_for.iterators |> erase(tab_index)
        new_for.iteratorsAt |> erase(tab_index)
        new_for.iteratorsAka |> erase(tab_index)
        new_for.iteratorsTags |> erase(tab_index)
        new_for.iteratorsTupleExpansion |> erase(tab_index)
        // Add keys(tab) and values(tab) as new sources
        new_for.sources |> emplace_new <| make_kv_call("keys", expr.sources[tab_index],␣
↪source_at)
        new_for.sources |> emplace_new <| make_kv_call("values", expr.sources[tab_index],
↪ source_at)
        // Add key and value iterator names
        let si = new_for.iterators |> length
        new_for.iterators |> resize(si + 2)
        new_for.iterators[si] := key_name
        new_for.iterators[si + 1] := val_name
        new_for.iteratorsAka |> resize(si + 2)
        new_for.iteratorsAka[si] := ""
```

(continues on next page)

(continued from previous page)

```
        new_for.iteratorsAka[si + 1] := ""
        new_for.iteratorsAt |> push(expr.iteratorsAt[tab_index])
        new_for.iteratorsAt |> push(expr.iteratorsAt[tab_index])
        new_for.iteratorsTags |> emplace_new <| clone_expression(expr.iteratorsTags[tab_
↪index])
        new_for.iteratorsTags |> emplace_new <| clone_expression(expr.iteratorsTags[tab_
↪index])
        new_for.iteratorsTupleExpansion |> push(0u8)
        new_for.iteratorsTupleExpansion |> push(0u8)
        // Clear iterator variables for re-inference
        new_for.iteratorVariables |> clear()
```

The helper `make_kv_call` builds an `ExprCall` node for `keys()` or `values()`:

```
def make_kv_call(fn_name : string; src_expr : ExpressionPtr; at : LineInfo) :␣
↪ExpressionPtr {
    var inscope call <- new ExprCall(at = at, name := fn_name)
    call.arguments |> emplace_new <| clone_expression(src_expr)
    return <- call
```

Using the macro

Section 1 — Verbose (without macro)

```
    print("--- Section 1: table iteration without the macro ---\n")
    var tab <- { "one" => 1, "two" => 2, "three" => 3 }
    for (k, v in keys(tab), values(tab)) {
        print("  {k} => {v}\n")
    }
}
```

This is the standard idiom: `keys()` and `values()` return separate iterators that advance in lock-step.

Section 2 — Tuple destructuring

```
    var tab <- { "one" => 1, "two" => 2, "three" => 3 }
    for ((k, v) in tab) {
        print("  {k} => {v}\n")
    }
```

Exactly the same output, but the syntax is more natural. The macro rewrites this to the verbose form transparently.

Section 3 — Mixed sources

The macro handles the table source independently; other sources in the same loop are preserved:

```
    var tab <- { "apple" => 10, "banana" => 20, "cherry" => 30 }
    for ((k, v), idx in tab, range(100)) {
        print("  [{idx}] {k} => {v}\n")
    }
```

Here `(k,v)` iterates over the table while `idx` counts from `range(100)`. All three advance in parallel.

Running the tutorial

```
daslang.exe tutorials/macros/09_for_loop_macro.das
```

Expected output:

```
--- Section 1: table iteration without the macro ---
  one => 1
  three => 3
  two => 2

--- Section 2: for ((k,v) in tab) with the macro ---
  one => 1
  three => 3
  two => 2

--- Section 3: mixed sources ---
  [0] apple => 10
  [1] banana => 20
  [2] cherry => 30
```

(Table iteration order may vary.)

Real-world example

The standard library module `daslib/soa.das` uses the same mechanism. Its `SoaForLoop` macro rewrites `for (it in soa_struct)` to iterate over the individual arrays of a structure-of-arrays layout — a more complex transformation but the same `AstForLoopMacro` pattern.

See also:

Full source: `09_for_loop_macro.das`, `for_loop_macro_mod.das`

Previous tutorial: tutorial_macro_variant_macro

Next tutorial: tutorial_macro_capture_macro

Standard library: `daslib/soa.das` (`SoaForLoop` macro)

Language reference: *Macros* — full macro system documentation

5.4.10 Macro Tutorial 10: Capture Macros

Previous tutorials transformed calls, functions, structures, blocks, variants, and for-loops. Capture macros intercept **lambda capture** — they fire when a lambda (or generator) captures outer variables, letting you wrap capture expressions, inject per-invocation cleanup, and add destruction-time release logic.

`[capture_macro(name="X")]` registers a class that extends `AstCaptureMacro`. The compiler calls three methods during lambda code generation:

`captureExpression(prog, mod, expr, etype)`
: Called **per captured variable** when the lambda struct is being built. `expr` is the expression being assigned to the capture field (typically an `ExprVar`). `etype` is the variable's type. Return a replacement expression to wrap the capture, or `default<ExpressionPtr>` to leave it unchanged.

`captureFunction(prog, mod, lcs, fun)`
: Called **once** after the lambda function is generated. `lcs` is the hidden lambda struct (with fields for each capture). `fun` is the lambda function. Use this to inspect captured fields and append code to `(fun.body as ExprBlock).finalList` — which runs **after each invocation** (per-call finally), not on destruction.

`releaseFunction(prog, mod, lcs, fun)`
: Called **once** when the lambda **finalizer** is generated. `fun` is the finalizer function (not the lambda call function). Code appended to `(fun.body as ExprBlock).list` runs on **destruction** — after the user-written `finally {}` block but before the compiler-generated field cleanup (`delete *__this`).

Note: Code added to `(fun.body as ExprBlock).finalList` by `captureFunction` runs after **every** lambda invocation. Code added to `(fun.body as ExprBlock).list` by `releaseFunction` runs **once** on destruction. The user-written `finally {}` on the lambda literal also runs on destruction (in the same finalizer), *before* `releaseFunction` code.

Generators are a special case — their function body's `finalList` runs on every yield iteration, which is why the standard library's `ChannelAndStatusCapture` skips generators entirely.

Motivation

When lambdas capture complex resources (file handles, GPU objects, reference-counted channels), it is useful to **audit** captures automatically — log when a resource is captured and verify it after each call — without modifying every lambda by hand.

This tutorial builds a capture macro driven by a **tag annotation**: only structs marked `[audited]` are monitored. Non-annotated types are silently ignored. This pattern (annotation + macro) is the same used by `daslib/jobque_boost.das` for `Channel` and `JobStatus` reference counting.

Note: Macros cannot be used in the module that defines them. This tutorial has **two** source files: a *module* file containing the macro definition and a *usage* file that requires the module.

The module file

`capture_macro_mod.das` defines four pieces:

1. `[audited]` — a no-op structure annotation used as a tag
2. Runtime helpers — `audit_on_capture`, `audit_after_invoke`, and `audit_on_finalize`
3. `CaptureAuditMacro` — the capture macro class (three hooks)

The tag annotation

```
[structure_macro(name=audited)]
class AuditedAnnotation : AstStructureAnnotation {
    def override apply(var st : StructurePtr; var group : ModuleGroup;
            args : AnnotationArgumentList; var errors : das_string) : bool {
        return true  // no-op tag
    }
}
```

This registers `[audited]` as a valid struct annotation. It does nothing at compile time — the capture macro checks for it at capture time.

Type checking helper

A `[macro_function]` inspects whether a `TypeDeclPtr` refers to a struct with the `[audited]` annotation:

```
[macro_function]
def private is_audited(typ : TypeDeclPtr) : bool {
    if (!typ.isStructure || typ.structType == null) {
        return false
    }
    for (ann in typ.structType.annotations) {
        if (ann.annotation.name == "audited") {
            return true
        }
    }
    return false
}
```

This iterates `typ.structType.annotations` — the same pattern used by `daslib/match.das` to check for `[match_as_is]`.

captureExpression

When an `[audited]` variable is captured, the macro wraps the capture expression in a call to `audit_on_capture(value, "name")`:

```
def override captureExpression(prog : Program?; mod : Module?;
        expr : ExpressionPtr; etype : TypeDeclPtr) : ExpressionPtr {
    if (!is_audited(etype)) {
        return <- default<ExpressionPtr>
    }
    var field_name = "unknown"
    if (expr is ExprVar) {
        field_name = string((expr as ExprVar).name)
    }
    var inscope pCall <- new ExprCall(at = expr.at,
        name := "capture_macro_mod::audit_on_capture")
    pCall.arguments |> emplace_new <| clone_expression(expr)
    pCall.arguments |> emplace_new <| new ExprConstString(
        at = expr.at, value := field_name)
    return <- pCall
}
```

`audit_on_capture` prints `[audit] captured 'name'` and returns the value unchanged, so the capture proceeds normally.

captureFunction

For each `[audited]` field in the lambda struct, the macro appends a print call to the function body's `finalList`:

```
def override captureFunction(prog : Program?; mod : Module?;
        var lcs : Structure?; var fun : FunctionPtr) : void {
    if (fun.flags._generator) {
        return    // generators run finally on every yield - skip
    }
    for (fld in lcs.fields) {
        if (!is_audited(fld._type)) {
            continue
        }
        if (true) {   // scope needed for var inscope inside a loop
            var inscope pCall <- new ExprCall(at = fld.at,
                name := "capture_macro_mod::audit_after_invoke")
            pCall.arguments |> emplace_new <| new ExprConstString(
                at = fld.at, value := string(fld.name))
            (fun.body as ExprBlock).finalList |> emplace(pCall)
        }
    }
}
```

The `if (true)` wrapper is required because `var inscope` is not allowed directly inside a `for` loop — the extra scope satisfies the compiler.

releaseFunction

For each `[audited]` field in the lambda struct, the macro appends a print call to the finalizer function's body — code that runs once on **destruction**, after the user-written `finally {}` block but before the compiler-generated `delete *__this`:

```
def override releaseFunction(prog : Program?; mod : Module?;
        var lcs : Structure?; var fun : FunctionPtr) : void {
    for (fld in lcs.fields) {
        if (!is_audited(fld._type)) {
            continue
        }
        if (true) {   // scope needed for var inscope inside a loop
            var inscope pCall <- new ExprCall(at = fld.at,
                name := "capture_macro_mod::audit_on_finalize")
            pCall.arguments |> emplace_new <| new ExprConstString(
                at = fld.at, value := string(fld.name))
            (fun.body as ExprBlock).list |> emplace(pCall)
        }
    }
}
```

Note that `releaseFunction` appends to `(fun.body as ExprBlock).list` (the finalizer body), **not** to `finalList`. The `fun` parameter here is the finalizer function — not the lambda call function received by `captureFunction`.

The finalizer execution order is:

1. User-written `finally {}` block (from the lambda literal)
2. `releaseFunction` code (this hook)
3. `delete *__this` — compiler-generated field destructors
4. `delete __this` — heap deallocation

The usage file

`10_capture_macro.das` defines an `[audited]` struct and a plain struct, then creates lambdas that capture them:

```
require capture_macro_mod

[audited]
struct Resource {
    name : string
    id : int
}

struct Plain {
    x : int
}
```

Section 1 — one `[audited]` and one plain capture:

```
var res = Resource(name = "texture.png", id = 1)
var pl = Plain(x = 42)
```

(continues on next page)

(continued from previous page)

```
var fn <- @() {
    print("  body: res.name={res.name}, pl.x={pl.x}\n")
}
fn()
unsafe { delete fn; }
```

Output:

```
[audit] captured 'res'
  body: res.name=texture.png, pl.x=42
[audit] after-call: 'res' still captured
  about to delete fn...
[audit] releasing 'res'
```

- `[audit] captured 'res'` — from `captureExpression` at lambda creation
- `[audit] after-call` — from `captureFunction`'s `finalList` after the call
- `[audit] releasing 'res'` — from `releaseFunction` during destruction
- No messages for `pl` (`Plain` has no `[audited]` annotation)

Section 2 — two `[audited]` captures, two calls:

```
var a = Resource(name = "mesh.obj", id = 2)
var b = Resource(name = "shader.hlsl", id = 3)
var fn <- @() {
    print("  body: a.id={a.id}, b.id={b.id}\n")
}
fn()
fn()
```

Each call produces after-call messages for both `a` and `b`. On destruction, releasing messages appear once for each field.

Section 3 — only non-annotated types (`int`): completely silent.

Real-world usage

The standard library's `ChannelAndStatusCapture` in `daslib/jobque_boost.das` uses the same hook pattern:

- `captureExpression`: calls `add_ref` on captured `Channel` or `JobStatus` pointers (increases reference count)
- `captureFunction`: appends a `panic` call that fires if the object was not properly released after each lambda invocation
- `releaseFunction`: could be used to call `release` on the captured object during destruction (complementing `add_ref`)

This ensures that thread-communication objects are never leaked, without requiring any changes to user lambda code.

See also:

Full source: `10_capture_macro.das`, `capture_macro_mod.das`

Previous tutorial: tutorial_macro_for_loop_macro

Next tutorial: tutorial_macro_reader_macro

Standard library: `daslib/jobque_boost.das` (`ChannelAndStatusCapture`)

Language reference: *Macros* — full macro system documentation

5.4.11 Macro Tutorial 11: Reader Macros

Previous tutorials intercepted calls, functions, structures, blocks, variants, for-loops, and lambda captures. Reader macros go further — they embed **entirely custom syntax** inside daslang source code.

`[reader_macro(name="X")]` registers a class that extends `AstReaderMacro`. Reader macros are invoked with the syntax `%X~ character_sequence %%`. The compiler calls three methods:

`accept(prog, mod, expr, ch, info)` → `bool`
Called **per character** during parsing. `ch` is the current character; `expr.sequence` accumulates the text. Return `true` to keep reading, `false` to stop (typically when `%%` is found).

`visit(prog, mod, expr)` → `ExpressionPtr`
Called during **type inference** on the `ExprReader` node. Must return an AST expression that replaces the reader expression. This is the **visit pattern** — used when the macro appears as an expression and produces an AST value.

`suffix(prog, mod, expr, info, outLine&, outFile?&)` → `string`
Called immediately **after** `accept()` during parsing. Returns a string of daslang source code that is injected back into the parser's input stream. This is the **suffix pattern** — used when the macro appears at module level and generates top-level daslang declarations (functions, structs, etc.). The `ExprReader` node is discarded.

Note: The two patterns serve different contexts:

- **Visit pattern**: the reader macro appears in an **expression** position (e.g., `var x = %csv~ ... %%`). `visit()` builds the resulting AST node. `suffix()` is not used.
- **Suffix pattern**: the reader macro appears at **module level** as a standalone statement (not assigned to a variable). `suffix()` returns daslang source text that the parser re-parses. `visit()` is never called because the parser discards the `ExprReader` node.

All reader macros share the same `accept()` idiom for collecting characters. The choice between visit and suffix determines where and how the macro produces output.

Motivation

Embedding domain-specific notations — CSV data, regular expressions, JSON literals, template engines — is a common need. Reader macros let you write these in their native syntax and transform them at compile time into efficient daslang code, without runtime parsing overhead.

This tutorial builds both patterns:

- `%csv~` — a **visit** reader macro that parses CSV text at compile time into a string array (constant embedded in the AST)
- `%basic~` — a **suffix** reader macro that transpiles a toy BASIC program into a daslang function definition

---

Note: Macros cannot be used in the module that defines them. This tutorial has **two** source files: a *module* file containing the macro definitions and a *usage* file that requires the module.

The module file

`reader_macro_mod.das` defines two reader macros.

The `accept()` idiom

Both macros share the same standard `accept()` implementation — the most common pattern in the standard library:

```
def override accept(prog : ProgramPtr; mod : Module?;
        var expr : ExprReader?; ch : int; info : LineInfo) : bool {
    if (ch != '\r') {               // skip carriage returns
        append(expr.sequence, ch)    // accumulate in expr.sequence
    }
    if (ends_with(expr.sequence, "%%")) {
        let len = length(expr.sequence)
        resize(expr.sequence, len - 2)   // strip the %%
        return false                     // stop reading
    } else {
        return true                      // keep reading
    }
}
```

Characters are appended to `expr.sequence` one at a time. When the `%%` terminator is detected, it is stripped and `accept()` returns `false` to signal the end of the character sequence.

CsvReader — visit pattern

`CsvReader` is registered with `[reader_macro(name=csv)]`. Its `visit()` method splits the collected sequence by commas, trims each value, and uses `convert_to_expression()` from `daslib/ast_boost` to embed the resulting string array in the AST:

```
def override visit(prog : ProgramPtr; mod : Module?;
        expr : smart_ptr<ExprReader>) : ExpressionPtr {
    if (is_in_completion()) {
        return <- default<ExpressionPtr>
    }
    let seq = string(expr.sequence)
    var items <- split(seq, ",")
    for (i in range(length(items))) {
        items[i] = strip(items[i])
    }
    return <- convert_to_expression(items, expr.at)
}
```

`convert_to_expression` takes any daslang value and converts it into AST nodes — here turning an `array<string>` into the equivalent of an array literal. This is the same utility used by `daslib/json_boost` to embed parsed JSON and by `daslib/regex_boost` to embed compiled regex objects.

BasicReader — suffix pattern

`BasicReader` is registered with `[reader_macro(name=basic)]`. It overrides `suffix()` instead of `visit()`. The method parses a tiny BASIC dialect and returns the equivalent daslang source code:

```
def override suffix(prog : ProgramPtr; mod : Module?;
        var expr : ExprReader?; info : LineInfo;
        var outLine : int&; var outFile : FileInfo?&) : string {
    let seq = string(expr.sequence)
    var lines <- split(seq, "\n")
    var func_name = "basic_program"
    var stmts : array<string>
    for (line in lines) {
        let trimmed = strip(line)
        if (empty(trimmed)) { continue }
        if (starts_with(trimmed, "DEF ")) {
            func_name = strip(slice(trimmed, 4))
            continue
        }
        // parse: NUMBER COMMAND args
        let sp1 = find(trimmed, " ")
        if (sp1 < 0) { continue }
        let after_num = strip(slice(trimmed, sp1 + 1))
        if (starts_with(after_num, "PRINT ")) {
            let arg = strip(slice(after_num, 6))
            if (starts_with(arg, "\"")) {
                let inner = slice(arg, 1, length(arg) - 1)
                stmts |> push("print(\"{inner}\\n\")")
            } else {
                stmts |> push("print(\"\{{arg}\}\\n\")")
            }
        } elif (starts_with(after_num, "LET ")) {
            stmts |> push("var {strip(slice(after_num, 4))}")
        }
    }
    var result = "def {func_name}() \{\n"
    for (stmt in stmts) {
        result += "    {stmt}\n"
    }
    result += "\}\n"
    return result
}
```

The returned string is valid gen2 daslang code. The parser receives this text and parses it as a normal function definition at module level.

Note: The suffix must produce **gen2 syntax** (with braces) if the source file uses `options gen2`. String escaping requires care: `\{` and `\}` produce literal braces (avoiding string interpolation), while `{func_name}` interpolates the variable.

The usage file

`11_reader_macro.das` demonstrates both patterns.

Section 1 — CSV reader (visit pattern):

```
var data <- %csv~ Alice, 30, New York %%
print("  items ({length(data)}):\n")
for (item in data) {
    print("    '{item}'\n")
}
```

Output:

```
--- Section 1: CSV reader (visit pattern) ---
  items (3):
    'Alice'
    '30'
    'New York'
  colors (3):
    'red'
    'green'
    'blue'
```

The `%csv~` expression evaluates to an `array<string>` — the CSV values are parsed and embedded at compile time, not at runtime.

Section 2 — BASIC transpiler (suffix pattern):

```
%basic~
DEF basic_hello
10 PRINT "Hello from BASIC"
20 LET x = 42
30 PRINT x
%%
```

This appears at **module level** (not inside a function). The suffix generates a function `basic_hello()` that the rest of the file can call:

```
[export]
def main() {
    basic_hello()
}
```

Output:

```
--- Section 2: BASIC transpiler (suffix pattern) ---
Hello from BASIC
42
```

The generated function is indistinguishable from hand-written code — it participates in type checking, AOT compilation, and all other compiler phases normally.

Visit vs Suffix

Aspect	Visit pattern	Suffix pattern
Override	`visit()`	`suffix()`
When called	Type inference	Parsing (after accept)
Returns	AST expression	daslang source text
Usage context	Expression position	Module level
ExprReader fate	Replaced by visit result	Discarded by parser
Examples (stdlib)	regex, json, stringify	spoof_instance

Real-world usage

The standard library includes several reader macros:

- `daslib/regex_boost.das` — `RegexReader` (visit): compiles a regular expression at parse time and embeds the compiled `Regex` struct directly in the AST. Usage: `%regex~pattern%%`
- `daslib/json_boost.das` — `JsonReader` (visit): parses JSON at compile time and embeds the resulting `JsonValue` tree. Usage: `%json~ {...} %%`
- `daslib/stringify.das` — `LongStringReader` (visit): embeds a multi-line string literal without escaping. Usage: `%stringify~ text %%`
- `daslib/spoof.das` — `SpoofTemplateReader` (visit) + `SpoofInstanceReader` (suffix): a template engine that stores templates as strings and instantiates them by generating daslang source code at parse time.

See also:

Full source: `11_reader_macro.das`, `reader_macro_mod.das`

Previous tutorial: tutorial_macro_capture_macro

Next tutorial: tutorial_macro_typeinfo_macro

Standard library: `daslib/regex_boost.das`, `daslib/json_boost.das`, `daslib/stringify.das`, `daslib/spoof.das`

Language reference: *Macros* — full macro system documentation

5.4.12 Macro Tutorial 12: Typeinfo Macros

Previous tutorials transformed calls, functions, structures, blocks, variants, for-loops, lambda captures, and reader macros. Typeinfo macros extend the built-in `typeinfo` expression with **custom compile-time type introspection**.

`[typeinfo_macro(name="X")]` registers a class that extends `AstTypeInfoMacro`. The macro is invoked with the syntax `typeinfo X(expr)` (gen2) or `typeinfo X<subtrait>(expr)`. During type inference the compiler calls:

`getAstChange(expr, errors)` → `ExpressionPtr`
: Receives an `ExprTypeInfo` node and returns a replacement AST expression — typically a constant (string, int, bool) or an array literal. The returned expression replaces the entire `typeinfo` call at compile time. Return `null` with an error message to report a compilation error.

ExprTypeInfo fields

The `expr` parameter exposes several fields:

Field	Type	Description
`typeexpr`	`TypeDeclPtr`	The type argument (from `type<T>`)
`subexpr`	`ExpressionPtr`	The expression argument (if used)
`subtrait`	`string`	The `<name>` in `typeinfo X<name>`
`extratrait`	`string`	The second `<a;b>` parameter
`trait`	`string`	The name of the typeinfo trait
`at`	`LineInfo`	Source location for error reporting

`typeexpr` gives access to the full type declaration. On structure types, `typeexpr.structType.fields` is a list of `FieldDeclaration` nodes (each with `name`, `_type`, `flags`). On enum types, `typeexpr.enumType.list` contains `EnumEntry` nodes (each with `name`).

Motivation

The `typeinfo` keyword already provides many built-in traits: `sizeof`, `typename`, `is_ref`, `has_field`, `is_enum`, `is_struct`, and dozens more. But sometimes you need *domain-specific* compile-time introspection — a description string for serialization, an array of enum names for UI dropdowns, or a boolean check for conditional compilation. `[typeinfo_macro]` lets you add such traits using pure daslang code.

The module file

The macro module defines three `AstTypeInfoMacro` subclasses, each returning a different expression type (string, array, bool).

Full source: `typeinfo_macro_mod.das`

struct_info — returning a string

`typeinfo struct_info(type<T>)` builds a description string at compile time:

```
[typeinfo_macro(name="struct_info")]
class TypeInfoGetStructInfo : AstTypeInfoMacro {
    def override getAstChange(expr : smart_ptr<ExprTypeInfo>;
                              var errors : das_string) : ExpressionPtr {
        if (expr.typeexpr == null) {
            errors := "type is missing or not inferred"
            return <- default<ExpressionPtr>
        }
        if (!expr.typeexpr.isStructure) {
            errors := "expecting structure type"
            return <- default<ExpressionPtr>
        }
        var result = "{expr.typeexpr.structType.name}("
        var first = true
        for (i in iter_range(expr.typeexpr.structType.fields)) {
            assume fld = expr.typeexpr.structType.fields[i]
```

(continues on next page)

(continued from previous page)

```
            if (fld.flags.classMethod) {
                continue
            }
            if (!first) {
                result += ", "
            }
            result += "{fld.name}:{describe(fld._type, false, false, false)}"
            first = false
        }
        result += ")"
        return <- new ExprConstString(at = expr.at, value := result)
    }
}
```

Key points:

- `expr.typeexpr.isStructure` validates that the type is a structure.
- `expr.typeexpr.structType.fields` iterates all field declarations.
- `fld.flags.classMethod` skips class methods (only data fields appear).
- `describe(fld._type, ...)` converts a `TypeDeclPtr` to a human-readable type name.
- The result is an `ExprConstString` — a compile-time string constant.

enum_value_strings — returning an array

`typeinfo enum_value_strings(type<E>)` returns a fixed-size array of enum value names:

```
[typeinfo_macro(name="enum_value_strings")]
class TypeInfoGetEnumValueStrings : AstTypeInfoMacro {
    def override getAstChange(expr : smart_ptr<ExprTypeInfo>;
                              var errors : das_string) : ExpressionPtr {
        // ... validation ...
        var inscope arr <- new ExprMakeArray(
            at = expr.at,
            makeType <- typeinfo ast_typedecl(type<string>))
        for (i in iter_range(expr.typeexpr.enumType.list)) {
            if (true) {
                assume entry = expr.typeexpr.enumType.list[i]
                var inscope nameExpr <- new ExprConstString(
                    at = expr.at, value := entry.name)
                arr.values |> emplace <| nameExpr
            }
        }
        return <- arr
    }
}
```

Key points:

- `ExprMakeArray` requires a `makeType` field — use `typeinfo ast_typedecl(type<string>)` to get the AST representation of `string`.
- `expr.typeexpr.enumType.list` iterates all `EnumEntry` nodes.

- The `if (true)` block is needed for `var inscope` lifetime scoping (each loop iteration creates and consumes a new `inscope` smart pointer).
- The result is a **fixed-size array** (`string[N]`), not a dynamic `array<string>`.

has_non_static_method — returning a bool with subtrait

`typeinfo has_non_static_method<name>(type<T>)` checks whether a class has a non-static method with the given name. The method name is passed via the `subtrait` parameter:

```
[typeinfo_macro(name="has_non_static_method")]
class TypeInfoHasNonStaticMethod : AstTypeInfoMacro {
    def override getAstChange(expr : smart_ptr<ExprTypeInfo>;
                              var errors : das_string) : ExpressionPtr {
        // ... validation ...
        if (empty(expr.subtrait)) {
            errors := "expecting method name as subtrait"
            return <- default<ExpressionPtr>
        }
        var found = false
        for (i in iter_range(expr.typeexpr.structType.fields)) {
            assume fld = expr.typeexpr.structType.fields[i]
            if (fld.name == expr.subtrait && fld.flags.classMethod) {
                found = true
                break
            }
        }
        return <- new ExprConstBool(at = expr.at, value = found)
    }
}
```

Key points:

- `expr.subtrait` is the string between angle brackets — in `typeinfo has_non_static_method<speak>(type<T>)`, `subtrait` is `"speak"`.
- Class methods are stored as struct fields with the `classMethod` flag.
- The result is `ExprConstBool` — a compile-time boolean, perfect for use in `static_if` conditional compilation.

The usage file

Full source: `12_typeinfo_macro.das`

Section 1: struct_info

```
struct Vec3 {
    x : float
    y : float
    z : float
}

struct Person {
```

(continues on next page)

(continued from previous page)

```
    name : string
    age : int
}

def section1() {
    let v = typeinfo struct_info(type<Vec3>)
    print("  Vec3: {v}\n")
    let p = typeinfo struct_info(type<Person>)
    print("  Person: {p}\n")
}
```

`typeinfo struct_info(type<Vec3>)` is replaced at compile time with the constant string `"Vec3(x:float, y:float, z:float)"`. No runtime reflection is involved.

Section 2: enum_value_strings

```
enum Color {
    Red
    Green
    Blue
}

def section2() {
    let colors = typeinfo enum_value_strings(type<Color>)
    for (c in colors) {
        print("    {c}\n")
    }
}
```

The macro generates a fixed-size `string[3]` array containing `"Red"`, `"Green"`, `"Blue"`. Use `let` (not `var <-`) to bind the result — fixed-size arrays are value types.

Section 3: has_non_static_method

```
class Animal {
    name : string
    def speak() {
        print("    {name} says hello\n")
    }
}

class Rock {
    weight : float
}

def section3() {
    let animal_can_speak = typeinfo has_non_static_method<speak>(type<Animal>)
    let rock_can_speak = typeinfo has_non_static_method<speak>(type<Rock>)
    static_if (typeinfo has_non_static_method<speak>(type<Animal>)) {
        print("  (static_if confirmed: Animal can speak)\n")
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The subtrait makes typeinfo macros parametric — the same macro handles any method name. Since the result is a compile-time `bool`, it works directly in `static_if` for conditional code generation.

Output

```
--- Section 1: struct_info ---
  Vec3: Vec3(x:float, y:float, z:float)
  Person: Person(name:string, age:int)
--- Section 2: enum_value_strings ---
  Color values (3):
    Red
    Green
    Blue
  Direction values (4):
    North
    South
    East
    West
--- Section 3: has_non_static_method ---
  Animal has 'speak': true
  Animal has 'fly': false
  Rock has 'speak': false
  (static_if confirmed: Animal can speak)
```

Real-world usage

The standard library provides several typeinfo macros:

- `typeinfo fields_count(type<T>)` — number of struct fields (`daslib/type_traits.das`)
- `typeinfo safe_has_property<name>(expr)` — does a type have a property function? (`daslib/type_traits.das`)
- `typeinfo enum_length(type<E>)` — number of enum values (`daslib/enum_trait.das`)
- `typeinfo enum_names(type<E>)` — array of enum value name strings (`daslib/enum_trait.das`)

See also:

Full source: `12_typeinfo_macro.das`, `typeinfo_macro_mod.das`

Previous tutorial: tutorial_macro_reader_macro

Next tutorial: tutorial_macro_enumeration_macro

Standard library: `daslib/type_traits.das`, `daslib/enum_trait.das`

Language reference: *Macros* — full macro system documentation

5.4.13 Macro Tutorial 13: Enumeration Macros

Previous tutorials transformed calls, functions, structures, blocks, variants, for-loops, lambda captures, reader macros, and typeinfo expressions. Enumeration macros let you intercept enum declarations at compile time and either **modify the enum** or **generate new code**.

`[enumeration_macro(name="X")]` registers a class that extends `AstEnumerationAnnotation`. This is the simplest macro type — it has only one method:

`apply(var enu, var group, args, var errors) → bool`
Called **before the infer pass**. The macro receives the full `EnumerationPtr` and can add/remove entries, generate functions, or create global variables. Return `true` on success, `false` with an error message to abort compilation.

Motivation

Enumerations in daslang are simple value lists. But real-world code often needs derived information:

- **A "total" sentinel** — the number of values, for array sizing or range checking.
- **String constructors** — converting user input strings to enum values at runtime.

Both can be achieved with enumeration macros:

- Section 1 shows `[enum_total]` — a custom macro that *modifies* an enum by appending a `total` entry.
- Section 2 shows `[string_to_enum]` from `daslib/enum_trait` — a standard library macro that *generates code* (a lookup table and two constructor functions).

The module file — enum_total

The macro module defines a single `AstEnumerationAnnotation` subclass that adds a `total` entry to any enum.

Full source: `enum_macro_mod.das`

```
[enumeration_macro(name="enum_total")]
class EnumTotalAnnotation : AstEnumerationAnnotation {
    def override apply(var enu : EnumerationPtr;
                       var group : ModuleGroup;
                       args : AnnotationArgumentList;
                       var errors : das_string) : bool {
        // Check that the enum doesn't already have a "total" entry.
        for (ee in enu.list) {
            if (ee.name == "total") {
                errors := "enumeration already has a 'total' field"
                return false
            }
        }
        // Add a new "total" entry at the end.
        let idx = add_enumeration_entry(enu, "total")
        if (idx < 0) {
            errors := "failed to add 'total' field"
            return false
        }
        // Set total = number of original entries.
        // length(enu.list) is now N+1, so total = N+1-1 = N.
        enu.list[idx].value |> move_new() <| new ExprConstInt(
```

(continues on next page)

(continued from previous page)

```
            at = enu.at, value = length(enu.list) - 1)
        return true
    }
}
```

Key points:

- `enu.list` is the array of `EnumEntry` nodes — iterate it to validate existing entries.
- `add_enumeration_entry(enu, "total")` appends a new entry and returns its index (or `-1` on failure).
- `enu.list[idx].value` is an `ExpressionPtr` — use `move_new` to assign a new `ExprConstInt` with the desired integer value.
- `enu.at` provides the source location for the generated expression.
- Return `false` with an error message to abort compilation — the error will point to the enum declaration.

The usage file

Full source: `13_enumeration_macro.das`

Section 1 — enum_total (modifying the enum)

```
require enum_macro_mod

[enum_total]
enum Direction {
    North
    South
    East
    West
}

def section1() {
    print("--- Section 1: enum_total ---\n")
    print("Direction.total = {int(Direction.total)}\n")
    for (d in each(Direction.North)) {
        if (d == Direction.total) {
            break
        }
        print("  {d}\n")
    }
}
```

The `[enum_total]` annotation runs before type inference and appends `total = 4` to the enum. At compile time the enum becomes:

```
enum Direction {
    North  = 0
    South  = 1
    East   = 2
    West   = 3
```

(continues on next page)

(continued from previous page)

```
    total  = 4
}
```

The `each()` function from `daslib/enum_trait` iterates all values (including `total`), so the loop breaks when it reaches the sentinel.

Section 2 — string_to_enum (generating code)

```
require daslib/enum_trait

[string_to_enum]
enum Color {
    Red
    Green
    Blue
}

def section2() {
    print("--- Section 2: string_to_enum ---\n")
    let c1 = Color("Red")
    print("Color(\"Red\")   = {c1}\n")
    let c2 = Color("invalid", Color.Blue)
    print("Color(\"invalid\", Color.Blue) = {c2}\n")
}
```

The `[string_to_enum]` annotation from `daslib/enum_trait` generates three things at compile time:

1. A **private global table** `_`enum`table`Color mapping strings to enum values — created with `add_global_private_let` and `enum_to_table`.
2. A **single-argument constructor** `Color(src : string) : Color` — panics if the string is not a valid enum name.
3. A **two-argument constructor** `Color(src : string; defaultValue : Color) : Color` — returns `defaultValue` if the string is not found.

Both constructors are generated with `qmacro_function`, registered with `add_function(compiling_module(), fn)`, and marked as non-private with `enumFn.flags &= ~FunctionFlags.privateFunction`.

How string_to_enum works internally

The `EnumFromStringConstruction` class in `daslib/enum_trait.das` demonstrates the **code generation** pattern for enumeration macros:

```
[enumeration_macro(name="string_to_enum")]
class EnumFromStringConstruction : AstEnumerationAnnotation {
    def override apply(var enu : EnumerationPtr;
                       var group : ModuleGroup;
                       args : AnnotationArgumentList;
                       var errors : das_string) : bool {
        var inscope enumT <- new TypeDecl(
            baseType = Type.tEnumeration,
```

(continues on next page)

(continued from previous page)

```
            enumType = enu.get_ptr())
        // 1. Create a private global lookup table.
        let varName = "_`enum`table`{enu.name}"
        add_global_private_let(
            compiling_module(), varName, enu.at,
            qmacro(enum_to_table(type<$t(enumT)>)))
        // 2. Generate the panic-on-miss constructor.
        var inscope enumFn <- qmacro_function("{enu.name}")
            $(src : string) : $t(enumT) {
                if (!key_exists($i(varName), src)) {
                    panic("enum value '{src}' not found")
                }
                return $i(varName)?[src] ?? default<$t(enumT)>
            }
        enumFn.flags &= ~FunctionFlags.privateFunction
        force_at(enumFn, enu.at)
        force_generated(enumFn, true)
        compiling_module() |> add_function(enumFn)
        // 3. Generate the default-on-miss constructor.
        var inscope enumFnDefault <- qmacro_function("{enu.name}")
            $(src : string; defaultValue : $t(enumT)) : $t(enumT) {
                return $i(varName)?[src] ?? defaultValue
            }
        enumFnDefault.flags &= ~FunctionFlags.privateFunction
        force_at(enumFnDefault, enu.at)
        force_generated(enumFnDefault, true)
        compiling_module() |> add_function(enumFnDefault)
        return true
    }
}
```

Key code-generation techniques:

- **``qmacro_function("name") $(args) : ReturnType { body }``** — creates a new function AST node. Reification splices (`$t()`, `$i()`) inject types and identifiers from variables.
- **``add_global_private_let(module, name, at, expr)``** — adds a private global `let` variable initialized by `expr`.
- **``compiling_module()``** — returns the module being compiled (where the annotated enum lives), so generated functions appear in the user's module.
- **``force_at(fn, at)``** — sets the source location of all nodes in the generated function to `at`, so error messages point to the enum declaration.
- **``force_generated(fn, true)``** — marks the function as compiler-generated (suppresses "unused function" warnings).

Output

```
--- Section 1: enum_total ---
Direction.total = 4
  North
  South
  East
  West
--- Section 2: string_to_enum ---
Color("Red")   = Red
Color("invalid", Color.Blue) = Blue
```

Real-world usage

`daslib/enum_trait` provides a rich set of enumeration utilities:

- `[string_to_enum]` — generates string constructors (shown above)
- `each(enumValue)` — iterates over all values of an enum type
- `string(enumValue)` — converts an enum value to its name
- `to_enum(type<E>, "name")` — runtime string-to-enum conversion
- `enum_to_table(type<E>)` — creates a `table<string; E>` lookup
- `typeinfo enum_length(type<E>)` — compile-time count of enum values
- `typeinfo enum_names(type<E>)` — compile-time array of value names

The two patterns shown in this tutorial cover the majority of enumeration macro use cases:

- **Modify the enum** — add sentinel values, computed entries, or validation (like `[enum_total]`).
- **Generate code** — create functions, tables, or variables derived from the enum's structure (like `[string_to_enum]`).

See also:

Full source: `13_enumeration_macro.das`, `enum_macro_mod.das`

Previous tutorial: tutorial_macro_typeinfo_macro

Next tutorial: tutorial_macro_pass_macro

Standard library: `daslib/enum_trait.das` — enum_trait module reference

Language reference: *Macros* — full macro system documentation

5.4.14 Macro Tutorial 14: Pass Macro

Previous tutorials showed macros attached to specific language constructs — calls, functions, structures, blocks, loops, enums, and typeinfo expressions. **Pass macros** operate at a higher level: they receive the **entire program** and run during a specific compilation phase.

`AstPassMacro` is the base class for all pass macros. It has a single method:

`apply(prog : ProgramPtr; mod : Module?) → bool`
`prog` is the full program being compiled. `mod` is the module that registered the macro. The return value depends on the annotation (see below).

Five annotations control **when** the macro runs:

Annotation	Behaviour
`[infer_macro]`	Runs after clean **type inference**. Returning `true` means "I changed something" — the compiler re-infers and runs macros again. The loop repeats until every macro returns `false`.
`[dirty_infer_macro]`	Runs during each **dirty inference** pass (the AST may be half-resolved). All dirty macros fire on every pass.
`[lint_macro]`	Invoked for each module compiled after the macro module, during the **lint phase**. Read-only — use it for analysis and diagnostics. Use `compiling_module()` to get the module currently being compiled.
`[global_lint_macro]`	Same as `[lint_macro]` but runs for **all** modules, not just the one that requires it.
`[optimization_macro]`	Runs during the **optimization** loop, after built-in optimisations. Returning `true` continues the loop.

This tutorial demonstrates the two most common types:

- **Section 1** — `[lint_macro]`: compile-time analysis (`CodeStatsLint`).
- **Section 2** — `[infer_macro]`: AST transformation (`TraceCallsPass`).

The module file

Full source: `pass_macro_mod.das`

Both macros live in a single module that the user requires.

Section 1 — lint_macro (compile-time analysis)

```
[lint_macro]
class CodeStatsLint : AstPassMacro {
    def override apply(prog : ProgramPtr; mod : Module?) : bool {
        let WARN_THRESHOLD = 4
        get_ptr(prog) |> for_each_module() $(var m : Module?) {
            if (m.moduleFlags.builtIn) {
                return  // skip C++ built-in modules
            }
            m |> for_each_function("") <| $(var func : FunctionPtr) {
                if (func.body == null || !(func.body is ExprBlock)) {
                    return
                }
                let body = func.body as ExprBlock
                let nStmts = length(body.list)
                if (nStmts > WARN_THRESHOLD) {
                    print("[lint] '{func.name}' has {nStmts} statements (>{WARN_
↪THRESHOLD})\n")
                }
            }
        }
        return false  // lint macros don't modify the AST
    }
}
```

Key points:

- `[lint_macro]` means this class runs **after inference succeeds**, during the read-only lint phase.
- `get_ptr(prog) |> for_each_module()` walks the **entire compiled program**. `m.moduleFlags.builtIn` skips C++ built-in modules so only user code is inspected.
- `for_each_function("")` iterates each module's functions. The empty string means "all names".
- The lint checks function body size: any function with more than `WARN_THRESHOLD` top-level statements triggers a compile-time warning via `print`.
- `print(...)` outputs at **compile time** — the message appears before any runtime output.
- The return value of a lint macro is ignored; lint macros never trigger re-inference.
- In production code, use `compiling_program() |> macro_error(at, text)` to emit **real compiler errors** instead of `print`. See `daslib/lint.das` for a full example.

Section 2 — infer_macro (AST transformation)

The infer macro instruments every function in the program by inserting a `_trace_enter("function_name")` call at the start of each function body. It follows the same visitor pattern as `daslib/heartbeat.das`.

First, a helper function that the injected code will call:

```
def public _trace_enter(name : string) {
    print(">>> {name}\n")
}
```

The visitor walks the AST and modifies function bodies:

```
class TraceCallsVisitor : AstVisitor {
    astChanged : bool = false
    @do_not_delete func : Function?
    def override preVisitFunction(var fun : FunctionPtr) {
        func = get_ptr(fun)
    }
    def override visitFunction(var fun : FunctionPtr) : FunctionPtr {
        // Skip our own helper to avoid infinite recursion at runtime.
        if (string(fun.name) == "_trace_enter") {
            func = null
            return <- fun
        }
        if (fun.body == null || !(fun.body is ExprBlock)) {
            func = null
            return <- fun
        }
        var body = fun.body as ExprBlock
        if (length(body.list) == 0) {
            func = null
            return <- fun
        }
        // Idempotency: skip if already instrumented.
        if ((body.list[0] is ExprCall) &&
            (body.list[0] as ExprCall).name == "_trace_enter") {
            func = null
```

(continues on next page)

(continued from previous page)

```
            return <- fun
        }
        // Insert _trace_enter("function_name") at the beginning.
        let fname = string(fun.name)
        var inscope expr <- qmacro(_trace_enter($v(fname)))
        body.list |> emplace(expr, 0)
        astChanged = true
        func.not_inferred()
        func = null
        return <- fun
    }
}
```

Key visitor techniques:

- ``**preVisitFunction**`` captures a raw pointer (`@do_not_delete func : Function?`) to the function being visited.
- ``**visitFunction**`` fires *after* the function body has been visited. It returns a `FunctionPtr` — returning `<- fun` keeps the original unchanged.
- **Self-exclusion** — `_trace_enter` must not instrument itself, or it would cause infinite recursion at runtime.
- **Idempotency** — checking whether the first statement is already a `_trace_enter` call prevents the macro from modifying the same function again on re-inference.
- ``**qmacro(_trace_enter($v(fname)))**`` generates a call expression. `$v(fname)` splices the string value of `fname` as a constant.
- ``**body.list |> emplace(expr, 0)**`` inserts the expression at position 0 (the start of the function body).
- ``**func.not_inferred()**`` marks the function as needing re-inference, so the compiler processes the injected call.
- ``**astChanged = true**`` signals that the pass made modifications.

The pass macro creates the visitor and walks the full program:

```
[infer_macro]
class TraceCallsPass : AstPassMacro {
    def override apply(prog : ProgramPtr; mod : Module?) : bool {
        var astVisitor = new TraceCallsVisitor()
        var inscope astVisitorAdapter <- make_visitor(*astVisitor)
        visit(prog, astVisitorAdapter)
        var result = astVisitor.astChanged
        unsafe {
            delete astVisitor
        }
        return result
    }
}
```

Key points:

- ``**new TraceCallsVisitor()**`` allocates the visitor on the heap (macro code cannot use stack-allocated visitors).
- ``**make_visitor(*astVisitor)**`` wraps it in a `smart_ptr` adapter for the `visit` function.
- ``**visit(prog, astVisitorAdapter)**`` walks the entire program — all modules, all functions. Use `visit(func, adapter)` to walk a single function instead.

- **Return value** — `true` tells the compiler to re-infer. Since the idempotency check prevents double-instrumentation, the second pass returns `false` and inference stabilises.
- ``**unsafe { delete astVisitor }**`` cleans up the raw pointer. The smart-ptr adapter is destroyed by `var inscope`.

The usage file

Full source: `14_pass_macro.das`

```
require pass_macro_mod

def greet(name : string) {
    print("Hello, {name}!\n")
}

def sum_to(n : int) : int {
    var total = 0
    for (i in range(n)) {
        total += i
    }
    return total
}

def countdown(n : int) {
    var i = n
    while (i > 0) {
        i--
    }
    print("counted down from {n}\n")
}

[export]
def main() {
    greet("world")
    let s = sum_to(5)
    print("sum = {s}\n")
    countdown(3)
}
```

The user code contains no trace calls — the `[infer_macro]` injects them automatically. The `[lint_macro]` prints its compile-time message before any runtime output.

Output

```
[lint] 'main' has 5 top-level statements (>4)
>>> main
>>> greet
Hello, world!
>>> sum_to
sum = 10
>>> countdown
counted down from 3
```

The first line is the lint macro's compile-time message — when linting the user module, it found that `main` has 5 top-level statements (the infer macro already added a `_trace_enter` call, raising its count above the threshold). The >>> lines come from the injected `_trace_enter` calls, proving that every user function was instrumented at compile time.

How it works — compilation pipeline

When the compiler processes the user's program:

1. **Parsing** — the source is parsed into an AST.
2. **Dirty inference** — `[dirty_infer_macro]` macros run on each pass (not used in this tutorial).
3. **Clean inference** — type inference runs. After it succeeds, `[infer_macro]` macros execute. `TraceCallsPass` instruments functions and returns `true`. The compiler re-infers. On the second pass, no new changes are made, so it returns `false`.
4. **Optimisation** — `[optimization_macro]` macros run in the optimisation loop (not used here).
5. **Lint** — `[lint_macro]` macros run for each module compiled after the macro module. `CodeStatsLint` inspects the user module via `compiling_module()` and warns about large function bodies. (`[global_lint_macro]` macros run once for the entire program.)
6. **Execution** — the instrumented program runs.

Avoiding infinite loops

An infer macro that always returns `true` will cause the compiler to hit its maximum pass limit and report an error. Two techniques prevent this:

- **Idempotency check** — before modifying a function, verify that the modification hasn't already been applied (e.g. check whether the first statement is already the injected call).
- **Targeted modification** — only modify functions that match specific criteria (e.g. skip `_trace_enter` to avoid self-instrumentation).

Real-world examples

The standard library uses pass macros for several purposes:

- **``daslib/heartbeat.das``** — `[infer_macro]` that inserts a `heartbeat()` call at every function and loop entry, enabling cooperative multitasking for long-running scripts.
- **``daslib/coverage.das``** — `[infer_macro]` that instruments every block with coverage-tracking calls.
- **``daslib/lint.das``** — `[lint_macro]` that enables paranoid compilation checks (unreachable code, unused variables, const upgrades, etc.).
- **``daslib/lint_everything.das``** — `[global_lint_macro]` that applies paranoid checks to **all** modules, not just the one that requires it.

See also:

Full source: `14_pass_macro.das`, `pass_macro_mod.das`

Previous tutorial: tutorial_macro_enumeration_macro

Next tutorial: tutorial_macro_type_macro

Standard library: `daslib/heartbeat.das`, `daslib/coverage.das`, `daslib/lint.das`

Language reference: *Macros* — full macro system documentation

5.4.15 Macro Tutorial 15: Type Macro

Previous tutorials showed macros attached to functions, structures, loops, enums, typeinfo expressions, and entire compilation passes. **Type macros** operate in a different domain: they let you define custom type expressions that the compiler resolves during type inference.

`AstTypeMacro` is the base class. It has a single method:

`visit(prog : ProgramPtr; mod : Module?; td : TypeDeclPtr; passT : TypeDeclPtr) → TypeDeclPtr`

`prog` is the program being compiled. `mod` is the module that registered the macro. `td` is the `TypeDecl` node representing the macro invocation — its `dimExpr` array carries the arguments. `passT` is non-null only in a generic context (see below). Return the resolved `TypeDeclPtr`, or an empty pointer on error.

The annotation `[type_macro(name="...")]` registers a class as a type macro. The name string determines the identifier used in type position:

```
[type_macro(name="padded")]
class PaddedTypeMacro : AstTypeMacro { ... }
```

After registration, `padded(type<float>, 4)` is a valid type expression.

How the parser stores arguments

When the compiler sees a type-macro invocation like `padded(type<float>, 4)` it creates a `TypeDecl` with `baseType = Type.typeMacro` and populates the `dimExpr` array:

Index	AST node	Contains
`dimExpr[0]`	`ExprConstString`	The macro name (`"padded"`)
`dimExpr[1]`	`ExprTypeDecl`	The first argument — the type (wraps `type<float>`)
`dimExpr[2]`	`ExprConstInt`	The second argument — the size (4)

Type arguments are wrapped in `ExprTypeDecl`; their inferred result is available via `dimExpr[i]._type`. Value arguments keep their expression type — `ExprConstInt` for integer literals, `ExprConstString` for strings, and so on.

Two inference contexts

The compiler calls `visit()` in two different contexts:

Concrete — all type arguments are already inferred.
Example: `var data : padded(type<float>, 4)`. Here `td.dimExpr[1]._type` is the fully-resolved `float` type, and `passT` is null. The macro clones `_type`, adds a dimension, and returns `float[4]`.

Generic — the type includes unresolved type parameters.
Example: `def foo(arr : padded(type<auto(TT)>, 4))`. Here `td.dimExpr[1]._type` is null because the type has not been inferred yet. `passT` is non-null and carries the actual argument type being matched. The macro must return a type the compiler can use for generic matching — typically by cloning the `ExprTypeDecl.typeexpr` (which preserves `auto(TT)`) and adding the dimension.

Important: Always check `td.dimExpr[i]._type == null` to distinguish the generic path from the concrete path. In the generic path, fall back to `.typeexpr` or create an `autoinfer` type.

Section 1 — The macro module

The type macro lives in its own module so that it is available via `require` — just like macros in previous tutorials.

Listing 1: tutorials/macros/type_macro_mod.das

```
options gen2
options no_aot

// Tutorial macro module: type macro (AstTypeMacro).
//
// AstTypeMacro lets you define custom type expressions that the compiler
// resolves during type inference.  It has a single method:
//   visit(prog : ProgramPtr; mod : Module?; td : TypeDeclPtr; passT : TypeDeclPtr) :␣
↪TypeDeclPtr
//
// When the compiler sees `padded(type<float>, 4)` in a type position it:
//   1. Parses it into a TypeDecl with baseType = Type.typeMacro
//   2. Stores the arguments in `td.dimExpr`:
//        td.dimExpr[0] - ExprConstString with the macro name ("padded")
//        td.dimExpr[1] - the type argument (ExprTypeDecl wrapping type<float>)
//        td.dimExpr[2] - the size argument (ExprConstInt with value 4)
//   3. Calls visit() so the macro can return the resolved type
//
// The `passT` parameter is non-null only in a generic context - it
// carries the actual argument type being matched against the parameter.
// In a concrete declaration (`var x : padded(type<float>, 4)`) passT
// is null and td.dimExpr[1]._type is already inferred.

module type_macro_mod

require ast
require daslib/ast_boost
require daslib/templates_boost

[type_macro(name="padded")]
class PaddedTypeMacro : AstTypeMacro {
    //! Type macro that resolves `padded(type<T>, N)` to `T[N]`.
    def override visit(prog : ProgramPtr; mod : Module?; td : TypeDeclPtr; passT :␣
↪TypeDeclPtr) : TypeDeclPtr {
        // --- argument validation ---
        // dimExpr must have exactly 3 entries: name + type + size.
        if (length(td.dimExpr) != 3) {
            macro_error(compiling_program(), td.at, "padded expects 2 arguments: type␣
↪and size")
            return <- TypeDeclPtr()
        }
        // The size argument must be a constant integer.
        if (!(td.dimExpr[2] is ExprConstInt)) {
            macro_error(compiling_program(), td.at, "padded: second argument must be a␣
↪constant integer")
            return <- TypeDeclPtr()
        }
```

(continues on next page)

(continued from previous page)

```
        let count = (td.dimExpr[2] as ExprConstInt).value

        // --- generic path ---
        // When the type argument is not yet inferred (e.g. in a generic
        // parameter like `padded(type<auto(TT)>, 4)`) we return a type
        // with autoinfer so the compiler can match and deduce TT.
        if (td.dimExpr[1]._type == null) {
            var inscope auto_type : TypeDeclPtr
            if (td.dimExpr[1] is ExprTypeDecl) {
                // Clone the unresolved type expression (e.g. auto(TT)).
                auto_type |> move_new <| clone_type((td.dimExpr[1] as ExprTypeDecl).
↪typeexpr)
            } else {
                // Fallback: pure auto-infer.
                auto_type |> move_new <| new TypeDecl(baseType = Type.autoinfer)
            }
            auto_type.dim |> push(count)
            return <- auto_type
        }

        // --- concrete path ---
        // The type argument is fully inferred.  Clone it and add the
        // dimension to produce the final array type (e.g. float[4]).
        var inscope final_type <- clone_type(td.dimExpr[1]._type)
        final_type.dim |> push(count)
        return <- final_type
    }
}
```

The `visit` method first validates that exactly two user arguments were provided (`dimExpr` length 3 — the name plus two arguments) and that the size argument is a constant integer.

For the **generic path** (`_type == null`): if the first argument is an `ExprTypeDecl` (e.g. wrapping `auto(TT)`), clone its `.typeexpr`; otherwise create a pure `autoinfer` type. Add the requested dimension and return.

For the **concrete path**: clone `_type` (the already-inferred type), add the dimension, and return the final type — e.g. `float[4]`.

Section 2 — Using the type macro

Listing 2: tutorials/macros/15_type_macro.das

```
options gen2
options no_aot

// Tutorial 15 - Type Macro (AstTypeMacro)
//
// This tutorial demonstrates how to create and use a custom type macro.
// The `padded` macro defined in type_macro_mod.das resolves the type
// expression `padded(type<T>, N)` into `T[N]`.
//
// Two usage patterns are shown:
```

(continues on next page)

(continued from previous page)

```
//   1. Concrete - fully specified types (`padded(type<float>, 4)`)
//   2. Generic  - auto-deduced types (`padded(type<auto(TT)>, 4)`)

require type_macro_mod

// --- Concrete usage ---
// The compiler sees `padded(type<float>, 4)` and calls PaddedTypeMacro.visit().
// The macro returns float[4], so `data` is a fixed-size array of 4 floats.

def test_concrete() {
    var data : padded(type<float>, 4)
    data[0] = 1.0
    data[1] = 2.0
    data[2] = 3.0
    data[3] = 4.0
    print("concrete: data = {data}\n")
}

// --- Generic usage ---
// When the type parameter uses `auto(TT)` the compiler resolves TT from
// the call site.  The macro returns auto(TT)[4] so the compiler can match
// and deduce the element type.

def sum_padded(arr : padded(type<auto(TT)>, 4)) : TT {
    var total : TT
    for (i in range(4)) {
        total += arr[i]
    }
    return total
}

[export]
def main() {
    test_concrete()

    // sum_padded deduces TT = float from the argument type.
    var floats : padded(type<float>, 4)
    floats[0] = 10.0
    floats[1] = 20.0
    floats[2] = 30.0
    floats[3] = 40.0
    let float_sum = sum_padded(floats)
    print("generic: float sum = {float_sum}\n")

    // sum_padded deduces TT = int from the argument type.
    var ints : padded(type<int>, 4)
    ints[0] = 1
    ints[1] = 2
    ints[2] = 3
    ints[3] = 4
    let int_sum = sum_padded(ints)
    print("generic: int sum = {int_sum}\n")
```

(continues on next page)

(continued from previous page)

```
}

// Expected output:
//   concrete: data = [[ 1; 2; 3; 4]]
//   generic: float sum = 100
//   generic: int sum = 10
```

`test_concrete` declares `var data : padded(type<float>, 4)`. The compiler invokes the macro in concrete mode — `_type` is `float`, so the macro returns `float[4]`. `data` is a fixed-size array of 4 floats.

`sum_padded` declares its parameter as `padded(type<auto(TT)>, 4)`. The compiler invokes the macro in generic mode. When called with `floats` (a `float[4]`), `TT` is deduced as `float`; when called with `ints` (an `int[4]`), `TT` is deduced as `int`.

Running the tutorial:

```
$ daslang tutorials/macros/15_type_macro.das
concrete: data = [[ 1; 2; 3; 4]]
generic: float sum = 100
generic: int sum = 10
```

See also:

Full source: `15_type_macro.das`, `type_macro_mod.das`

Previous tutorial: tutorial_macro_pass_macro

Next tutorial: tutorial_macro_template_type_macro

Language reference: *Macros* — full macro system documentation

5.4.16 Macro Tutorial 16: Template Type Macro

Tutorial 15 showed a type macro that returns an array type. This tutorial goes further — the `pair` macro **generates a structure** with type-parameterized fields, turning `pair(type<int>, type<float>)` into a struct with `first : int` and `second : float`.

The implementation uses the *manual* approach (without the `[template_structure]` shorthand) to expose the mechanics behind template type resolution: structure aliases, generic inference, and template class instantiation.

Key concepts

Structure aliases — every generated struct stores its type parameters as aliases (`T1` → `int`, `T2` → `float`). On the generic path the compiler reads them back to deduce `auto(T1)`, `auto(T2)`.

``**[typemacro_function]**`` annotation (from `daslib/typemacro_boost`) — converts a regular function into an `AstTypeMacro`. It auto-generates the class, registers it with `add_new_type_macro`, and extracts `dimExpr` arguments into the function parameters. The first two parameters are always `macroArgument` (the `TypeDecl` node) and `passArgument` (non-null in generic context); subsequent parameters are the user arguments.

``**TypeMacroTemplateArgument**`` — a helper struct that pairs a template parameter name with its declared type (`argument_type`) and, after inference, the resolved concrete type (`inferred_type`).

Template definition

The template struct uses alias names (`T1`, `T2`) for its fields. The `template` keyword prevents direct instantiation — these aliases only make sense after the type macro resolves them:

```
struct template Pair {
    first  : T1
    second : T2
}
```

Two paths — concrete and generic

The type macro function handles two compiler contexts:

Concrete (`passArgument == null`) — the user wrote fully-specified types like `pair(type<int>, type<float>)`:

```
1. verify_arguments     - ensure no remaining auto types
2. template_structure_name - build mangled name "Pair<int,float>"
3. find_unique_structure   - deduplicate (skip if already exists)
4. qmacro_template_class  - clone the template struct, rename
   it, clear the template flag, apply substitution rules
5. make_typemacro_template_instance - annotate the new struct so
   it can be recognized as a Pair instance later
6. add_structure_aliases - store T1=int, T2=float on the struct
```

Generic (`passArgument != null`) — a function parameter uses `auto(...)` patterns like `pair(type<auto(T1)>, type<auto(T2)>)` and the compiler is matching a concrete argument against it:

```
1. is_typemacro_template_instance - verify the argument is indeed
   a Pair instance
2. infer_struct_aliases - read T1, T2 aliases from the concrete
   struct into template_arguments[i].inferred_type
3. infer_template_types - for each argument, call
   infer_generic_type to match auto(T1) against the actual type,
   then update_alias_map so the compiler resolves the names
```

Section 1 — The macro module

Listing 3: tutorials/macros/template_type_macro_mod.das

```
options gen2
options no_aot

// Tutorial macro module: template type macro (structure generation).
//
// This module defines a `pair` type macro that produces a structure
// with two fields - `first : T1` and `second : T2`.  The macro is
// implemented manually (without the `[template_structure]` shorthand)
// to show the mechanics behind template type resolution:
//
//   1. `[typemacro_function]` annotation converts a regular function
```

(continues on next page)

(continued from previous page)

```
//      into an AstTypeMacro.  It also auto-extracts the dimExpr
//      arguments into the function parameters.
//
//   2. On the **concrete** path (`passArgument == null`) the macro
//      generates a new structure via `qmacro_template_class`, gives
//      it a mangled name like `Pair<int,float>`, and stores each type
//      parameter as a **structure alias** - these aliases tell the
//      compiler how to resolve `T1` and `T2` inside the cloned struct.
//
//   3. On the **generic** path (`passArgument != null`) the macro
//      reads the aliases back from the concrete struct that was passed
//      as the argument, then uses `infer_template_types` to match
//      `auto(T1)` / `auto(T2)` against the actual alias values and
//      update the compiler's alias map.

module template_type_macro_mod

require daslib/typemacro_boost

// ---------------------------------------------------------------------------
// Template definition.
//
// `struct template` prevents direct instantiation - fields use alias
// names (T1, T2) that only make sense after template substitution.
// ---------------------------------------------------------------------------

struct template Pair {
    first  : T1
    second : T2
}

// ---------------------------------------------------------------------------
// Type macro function.
//
// `[typemacro_function]` generates the AstTypeMacro boilerplate:
//   • Creates a class derived from AstTypeMacro
//   • Registers it with add_new_type_macro(name="pair")
//   • Auto-extracts dimExpr arguments into function parameters
//     (macroArgument = td, passArgument = passT, then user args)
// ---------------------------------------------------------------------------

[typemacro_function]
def pair(macroArgument, passArgument : TypeDeclPtr; T1, T2 : TypeDeclPtr) : TypeDeclPtr {
    // Get the TypeDecl that points to our template struct.
    var inscope template_type <- typeinfo ast_typedecl(type<Pair>)

    // Describe the template arguments - names must match the alias
    // names used in the struct fields (T1, T2).
    var inscope template_arguments <- [
        TypeMacroTemplateArgument(name = "T1", argument_type <- clone_type(T1)),
        TypeMacroTemplateArgument(name = "T2", argument_type <- clone_type(T2))
    ]
```

(continues on next page)

(continued from previous page)

```
    // No extra (non-type) arguments for this macro.
    var inscope extra_template_arguments : array<tuple<string; string>>

    // --- generic path ---
    // The compiler passes a concrete type it wants to match against
    // our template signature.  For example, when a function parameter
    // is `pair(type<auto(A)>, type<auto(B)>)` and the caller provides
    // a `Pair<int,float>`, passArgument is the Pair<int,float> type.
    if (passArgument != null) {
        // Verify the concrete type is an instance of our Pair template.
        if (!is_typemacro_template_instance(passArgument, template_type, extra_template_
↪arguments)) {
            return <- TypeDeclPtr()
        }
        // Read the stored aliases (T1, T2) from the concrete struct
        // back into template_arguments[i].inferred_type.
        if (!infer_struct_aliases(passArgument.structType, template_arguments)) {
            return <- TypeDeclPtr()
        }
        // Match the inferred concrete types against the template
        // parameters (which may contain auto(...) patterns) and
        // update the compiler's alias map for further resolution.
        return <- infer_template_types(passArgument, template_arguments)
    }

    // --- concrete path ---
    // passArgument is null - the user wrote a concrete type like
    // `pair(type<int>, type<float>)`.  We need to generate (or
    // look up) the corresponding structure.

    // All type arguments must be fully resolved (no remaining auto).
    if (!verify_arguments(template_arguments)) {
        return <- TypeDeclPtr()
    }

    // Build a mangled name like "Pair<int,float>" for deduplication.
    var struct_name = template_structure_name(template_type.structType, template_
↪arguments, extra_template_arguments)

    // If this exact instantiation already exists, reuse it.
    var existing_struct = compiling_program().find_unique_structure(struct_name)
    if (existing_struct != null) {
        return <- new TypeDecl(baseType = Type.tStructure, structType = existing_struct,␣
↪at = template_type.at)
    }

    // Clone the template struct, rename it, clear the template flag,
    // and apply substitution rules so field types resolve correctly.
    var inscope resType <- qmacro_template_class(struct_name, type<Pair>)

    // Annotate the new struct so `is_typemacro_template_instance`
```

(continues on next page)

(continued from previous page)

```
    // can identify it as an instance of Pair later (for generics).
    make_typemacro_template_instance(resType.structType, template_type.structType, extra_
↪template_arguments)

    // Store T1 and T2 as structure aliases - this is how
    // `infer_struct_aliases` will read them back on the generic path.
    add_structure_aliases(resType.structType, template_arguments)

    return <- resType
}
```

The `[typemacro_function]` annotation on `pair` converts the function into an `AstTypeMacro` registered under the name `"pair"`. The annotation also generates code that extracts `dimExpr[1]` and `dimExpr[2]` into the `T1` and `T2` `TypeDeclPtr` parameters.

The argument names in `TypeMacroTemplateArgument` (`"T1"`, `"T2"`) **must** match the alias names used in the template struct fields. This is how `qmacro_template_class` knows which alias to substitute when cloning the structure.

Section 2 — Using the type macro

Listing 4: tutorials/macros/16_template_type_macro.das

```
options gen2
options no_aot

// Tutorial 16 - Template Type Macro (structure generation)
//
// This tutorial uses the `pair` type macro from template_type_macro_mod
// to create parameterized structures.  `pair(type<T1>, type<T2>)` expands
// to a struct with `first : T1` and `second : T2`.
//
// Three patterns are demonstrated:
//   1. Concrete instantiation - fully specified types
//   2. Generic parameter     - auto-deduced type arguments
//   3. Generic return type   - the macro in return-type position

require template_type_macro_mod

// --- 1. Concrete instantiation ---
// `pair(type<int>, type<float>)` generates a struct `Pair<int,float>`
// with `first : int` and `second : float`.

def test_concrete() {
    var p : pair(type<int>, type<float>)
    p.first = 42
    p.second = 3.14
    print("concrete: first={p.first} second={p.second}\n")
}

// --- 2. Generic parameter ---
// The function accepts any Pair instantiation.  The compiler matches
// `auto(T1)` / `auto(T2)` against the stored structure aliases.
```

(continues on next page)

(continued from previous page)

```
def get_first(p : pair(type<auto(T1)>, type<auto(T2)>)) : T1 {
    return p.first
}

def get_second(p : pair(type<auto(T1)>, type<auto(T2)>)) : T2 {
    return p.second
}

// --- 3. Generic return type ---
// The macro can appear in return-type position as well.  Here we swap
// the type arguments - `pair(type<T2>, type<T1>)` - so the returned
// struct has the types reversed.

def swap_pair(p : pair(type<auto(T1)>, type<auto(T2)>)) : pair(type<T2>, type<T1>) {
    var result : pair(type<T2>, type<T1>)
    result.first = p.second
    result.second = p.first
    return result
}

[export]
def main() {
    test_concrete()

    // Generic parameter: get_first / get_second.
    var p : pair(type<int>, type<float>)
    p.first = 10
    p.second = 2.5
    print("get_first:  {get_first(p)}\n")
    print("get_second: {get_second(p)}\n")

    // Generic return type: swap_pair.
    var swapped = swap_pair(p)
    print("swapped: first={swapped.first} second={swapped.second}\n")
}

// Expected output:
//   concrete: first=42 second=3.14
//   get_first:  10
//   get_second: 2.5
//   swapped: first=2.5 second=10
```

Three patterns are demonstrated:

1. **Concrete instantiation** — `var p : pair(type<int>, type<float>)` triggers the concrete path. The macro generates a struct `Pair<int,float>` with fields `first : int` and `second : float`.
2. **Generic parameter** — `get_first` and `get_second` accept any `Pair` instantiation. The compiler matches `auto(T1)` / `auto(T2)` against the stored aliases and deduces the element types.
3. **Generic return type** — `swap_pair` returns `pair(type<T2>, type<T1>)` — the swapped types. Both generic parameter inference and concrete struct generation happen in the same call.

Running the tutorial:

```
$ daslang tutorials/macros/16_template_type_macro.das
concrete: first=42 second=3.14
get_first:  10
get_second: 2.5
swapped: first=2.5 second=10
```

See also:

Full source: `16_template_type_macro.das`, `template_type_macro_mod.das`

Previous tutorial: tutorial_macro_type_macro

Next tutorial: tutorial_macro_qmacro

Standard library: `daslib/typemacro_boost.das` — infrastructure for template type macros (`TypeMacroTemplateArgument`, `infer_template_types`, `add_structure_aliases`, etc.)

Language reference: *Macros* — full macro system documentation

5.4.17 Macro Tutorial 17: Quasi-quotation Reference

Tutorial 16 used `qmacro_template_class` to generate a struct from a template. This tutorial is a **comprehensive reference** — it exercises every quasi-quotation variant and reification operator provided by `daslib/templates_boost` in a single macro module.

Quasi-quotation variants

All functions below are `[call_macro]` helpers (or the underlying runtime functions) from `templates_boost`. They build AST nodes at macro expansion time:

Variant	Description
`quote(expr)`	Plain AST capture, **no** reification. Use for literal constants.
`qmacro(expr)`	Expression with reification splices (`$v`, `$e`, etc.).
`qmacro_type(type<T>)`	Build a `TypeDeclPtr`. `$t(td)` splices another `TypeDeclPtr` inside.
`qmacro_variable("name", type<T>)`	Build a `VariablePtr`. Used for function parameters.
`qmacro_expr(${ stmt; })`	Single statement (e.g. `let` / `var` declaration).
`qmacro_block() { stmts }`	Multiple statements as an `ExprBlock`. `$b(stmts)` splices an array.
`qmacro_block_to_array() { stmts }`	Like `qmacro_block` but returns `array<ExpressionPtr>`.
`qmacro_function("name") $(args) { body }`	Complete function. `$a(args)` splices the parameter list.
`qmacro_template_class("Name", type<T>)`	Clone a `struct template`, rename it, apply substitution rules. Also clones and renames any methods defined on the template.
`qmacro_method("Cls\`name", cls) $(self...) { }`	Class method. Used here to add `get_tuple` to the cloned struct.
`qmacro_template_function(@@Th...)`	Clone a `def template` function. Types fixed up manually.

Reification operators

Inside `qmacro(...)` and friends, dollar-prefixed splices inject compile-time values into the generated AST:

Operator	Meaning	Example
`$v(val)`	Compile-time **value** → AST constant	`$v("hello")` → `ExprConstString`
`$e(expr)`	Splice an existing `ExpressionPtr`	`$e(existing_ast)` → inserts that node
`$i(name)`	String → **identifier** reference	`$i("tag")` → `ExprVar { name="tag" }`
`$c(name)`	String → function **call** name	`$c("print")` → `ExprCall { name="print" }`
`$f(name)`	String → **field** access name	`pair.$f("first")` → `pair.first`
`$t(td)`	Splice a `TypeDeclPtr` in type position	`type<array<$t(int_type)>>` → `type<array<int>>`
`$a(args)`	Splice `array<VariablePtr>` as parameters	`$($a(fn_args))` in `qmacro_function`
`$b(body)`	Splice `array<ExpressionPtr>` as block body	`{ $b(fn_body) }` in `qmacro_function`

Section 1 — The macro module

The entire macro is a single `[call_macro]` named `build_demo`. Its `visit` method exercises **every** variant listed above, including `qmacro_method` — used to add a `get_tuple` method to the cloned `DemoIntFloat` struct.

Listing 5: tutorials/macros/qmacro_mod.das

```
options gen2
options no_aot

// Tutorial macro module: quasi-quotation reference (qmacro variants).
//
// This module demonstrates every qmacro variant and reification
// operator through a single [call_macro] named `build_demo`.
// When the user writes `build_demo()` the macro generates:
//
//   • A struct via qmacro_template_class  - DemoIntFloat (with method)
//   • A function via qmacro_function      - demo_run
//   • A function clone via qmacro_template_function - demo_add_int
//
// Every other qmacro variant (qmacro, qmacro_block, qmacro_block_to_array,
// qmacro_expr, qmacro_type, qmacro_variable, quote) and every reification
// splice ($v, $e, $i, $c, $f, $t, $a, $b) is exercised in the process.
//
// Every variant is exercised, including qmacro_method - used here
// to generate a `get_tuple` method on the cloned struct.

module qmacro_mod public

require ast
require daslib/ast_boost
require daslib/templates_boost

// ---------------------------------------------------------------------
// Template struct - used by qmacro_template_class.
```

(continues on next page)

(continued from previous page)

```
// `struct template` prevents direct instantiation.  TFirst / TSecond
// are alias names resolved when the template is cloned.
//
// The describe() method is defined inside the struct body.  When
// qmacro_template_class clones DemoPair into DemoIntFloat it also
// clones this method, renames it to DemoIntFloat`describe, and applies
// the same type substitution rules (TFirst → int, TSecond → float) to
// the method's self parameter and body.
// ---------------------------------------------------------------------
struct template DemoPair {
    first  : TFirst
    second : TSecond

    def describe() {
        print("first  = {self.first}\n")
        print("second = {self.second}\n")
    }
}

// ---------------------------------------------------------------------
// Template function - used by qmacro_template_function.
// The $t(t_value) tag types in the signature are resolved automatically
// when qmacro_template_function clones the function - it reads the tag
// and substitutes whatever TypeDeclPtr `t_value` holds at expansion time.
// ---------------------------------------------------------------------
def template demo_add(a, b : $t(t_value)) : $t(t_value) {
    return a + b
}

// ---------------------------------------------------------------------
// The main call macro.
// ---------------------------------------------------------------------

[call_macro(name="build_demo")]
class BuildDemoMacro : AstCallMacro {
    def override visit(prog : ProgramPtr; mod : Module?; var expr : smart_ptr
↪<ExprCallMacro>) : ExpressionPtr {

        // ============================================================
        // 1. quote(expr) - plain AST capture, NO reification.
        //    Returns ExpressionPtr.  Use for simple literal constants.
        // ============================================================
        var inscope literal_true <- quote(true)   // ExprConstBool
        var inscope literal_zero <- quote(0)      // ExprConstInt

        // ============================================================
        // 2. qmacro(expr) - expression with reification splices.
        //    Returns ExpressionPtr.
        //
        //    Reification operators used here:
        //      $v(value) - splice compile-time VALUE as a constant
        //      $e(expr)  - splice an existing ExpressionPtr
```

(continues on next page)

(continued from previous page)

```
// ============================================================
let greeting = "hello from qmacro"

// $v - compile-time string becomes a constant in the AST.
var inscope print_greeting <- qmacro(print($v("{greeting}!\n")))

// $e - insert an already-built ExpressionPtr into the tree.
var inscope msg_expr <- qmacro("the literal is: ")
var inscope print_literal <- qmacro(print($e(msg_expr) + "{$e(literal_true)}\n"))

// ============================================================
// 3. qmacro_type(type<T>) - build a TypeDeclPtr.
//    $t(typeDecl) - splice a TypeDeclPtr in type position.
// ============================================================
var inscope int_type   <- qmacro_type(type<int>)
var inscope float_type <- qmacro_type(type<float>)
// $t - compose types: array<$t(int_type)> → array<int>
var inscope arr_int_type <- qmacro_type(type<array<$t(int_type)>>)

// ============================================================
// 4. qmacro_variable("name", type<T>) - build a VariablePtr.
//    Used to create function arguments programmatically.
// ============================================================
var inscope tag_var <- qmacro_variable("tag", type<string>)

// ============================================================
// 5. qmacro_expr(${ statement; }) - single statement.
//    Returns ExpressionPtr.  Useful for let/var declarations
//    that need statement syntax in an expression context.
// ============================================================
var inscope let_stmt <- qmacro_expr(${ let title = "items"; })

// ============================================================
// 6. qmacro_block() { stmts } - multiple statements as ExprBlock.
//    $b(stmts) - splice array<ExpressionPtr> as a block body.
// ============================================================
var inscope steps : array<ExpressionPtr>
steps |> emplace_new <| qmacro(print("  step 1\n"))
steps |> emplace_new <| qmacro(print("  step 2\n"))
steps |> emplace_new <| qmacro(print("  step 3\n"))

var inscope block_stmts <- qmacro_block() {
    print("qmacro_block body:\n")
    $b(steps)
}

// ============================================================
// 7. qmacro_block_to_array() { stmts }
//    Like qmacro_block but returns array<ExpressionPtr> -
//    each statement becomes a separate element.
// ============================================================
var inscope extra <- qmacro_block_to_array() {
```

(continues on next page)

(continued from previous page)

```
            print("  extra A\n")
            print("  extra B\n")
        }

        // ============================================================
        // 8. qmacro_function("name") $(args) : RetType { body }
        //    Generates a complete FunctionPtr.
        //
        //    Reification operators used here:
        //      $a(args) - splice array<VariablePtr> as parameter list
        //      $b(body) - splice array<ExpressionPtr> as function body
        //      $i(name) - splice a string as an IDENTIFIER reference
        //      $c(name) - splice a string as a function CALL name
        //      $f(name) - splice a string as a field access name
        // ============================================================
        let fn_name = "demo_run"
        var inscope fn_args : array<VariablePtr>
        fn_args |> emplace_new <| clone_variable(tag_var)

        // Build the function body - each statement demonstrates
        // a different reification operator or qmacro variant.
        var inscope fn_body : array<ExpressionPtr>

        // $v - greeting is baked in as a string constant.
        fn_body |> emplace_new <| clone_expression(print_greeting)

        // $e - two sub-expressions spliced into a larger expression.
        fn_body |> emplace_new <| clone_expression(print_literal)

        // $c - string becomes a function call name.
        //      $c("print") → ExprCall { name = "print" }
        let print_fn = "print"
        fn_body |> emplace_new <| qmacro($c(print_fn)($v("called via $c\n")))

        // $i - string becomes an identifier (variable reference).
        //      $i("tag") → ExprVar { name = "tag" }
        let tag_name = "tag"
        fn_body |> emplace_new <| qmacro(print("$i resolved tag = {$i(tag_name)}\n"))

        // $f - string becomes a field access name.
        //      pair.$f("first") → pair.first
        let field_name = "first"
        fn_body |> emplace_new <| qmacro_expr($${
            var pair = DemoIntFloat(first = 42, second = 3.14);
        })
        fn_body |> emplace_new <| qmacro(print("pair.$f first = {pair.$f(field_name)}\n
↪"))

        // qmacro_block result - three steps spliced via $b.
        fn_body |> emplace_new <| clone_expression(block_stmts)

        // qmacro_block_to_array - extra statements appended.
```

(continues on next page)

(continued from previous page)

```
    for (s in extra) {
        fn_body |> emplace_new <| clone_expression(s)
    }

    // qmacro_expr - a let declaration as a single statement.
    fn_body |> emplace_new <| clone_expression(let_stmt)
    fn_body |> emplace_new <| qmacro(print("title = {title}\n"))

    // $a - splice argument list; $b - splice body statements.
    var inscope demo_fn <- qmacro_function(fn_name) $($a(fn_args)) {
        $b(fn_body)
    }
    demo_fn.flags |= FunctionFlags.generated
    add_function(compiling_module(), demo_fn)

    // ============================================================
    // 9. qmacro_template_class("Name", type<Template>)
    //    Clones a struct template, renames it, clears the template
    //    flag, and returns a TypeDeclPtr pointing to the new struct.
    //
    //    Type aliases are resolved via add_structure_alias - this
    //    registers alias types on the cloned struct so the compiler
    //    replaces TFirst/TSecond with real types during compilation.
    //
    //    Also clones DemoPair`describe → DemoIntFloat`describe,
    //    applying the same alias substitution to the method body.
    // ============================================================
    var inscope pair_type <- qmacro_template_class("DemoIntFloat", type<DemoPair>)
    pair_type.structType |> add_structure_alias("TFirst", int_type)
    pair_type.structType |> add_structure_alias("TSecond", float_type)

    // ============================================================
    // 10. qmacro_template_function(@@template_fn)
    //     Clones a template function, strips the template flag.
    //     Takes a function address (@@), not a string name.
    //     The template uses $t(t_value) tag types in its signature,
    //     so qmacro_template_function auto-generates substitution
    //     rules - no manual type fixup is needed.
    // ============================================================
    var inscope t_value <- clone_type(int_type)
    var inscope add_fn <- qmacro_template_function(@@demo_add)
    add_fn.name := "demo_add_int"
    add_fn.flags |= FunctionFlags.generated
    add_function(compiling_module(), add_fn)

    // ============================================================
    // 11. qmacro_method("Cls`name", cls) $(var self : T) { body }
    //     Generates a FunctionPtr marked as a class method.
    //     Here we add a `get_tuple` method to DemoIntFloat that
    //     returns (first, second) as a tuple.
    // ============================================================
```

(continues on next page)

(continued from previous page)

```
        var inscope pair_struct_type <- add_ptr_ref(pair_type.structType)
        var inscope method <- qmacro_method("get_tuple", pair_struct_type) $(var self :
↪$t(pair_type)) {
            return (self.first, self.second)
        }
        add_function(compiling_module(), method)

        // ============================================================
        // Return value - all real work was done via add_function /
        // qmacro_template_class.  Return a trivial expression.
        // ============================================================
        return <- quote(true)
    }
}
```

Key observations:

- **``quote`` vs ``qmacro``** — use `quote` when no `$...` splices are needed; use `qmacro` when you need reification.
- **``qmacro_function`` + ``$a`` + ``$b``** — builds a complete function from dynamically constructed argument and body arrays.
- **``qmacro_template_class`` + ``add_structure_alias``** — the call macro clones the `struct template`, renames it, and returns a `TypeDeclPtr`. `add_structure_alias` then registers alias types (`TFirst` → `int`, `TSecond` → `float`) on the cloned struct so the compiler resolves them during compilation. The template's `describe()` method is also cloned and renamed to `DemoIntFloat`describe` with the same alias substitution applied.
- **``qmacro_method``** — generates a standalone method (`get_tuple`) and attaches it to the cloned `DemoIntFloat` struct. Uses `add_ptr_ref` to obtain the `StructurePtr` and `$t(pair_type)` to splice the struct type into the `self` parameter.
- **``qmacro_template_function``** — clones a `def template` function. The template uses `$t(t_value)` tag types in its signature, so `qmacro_template_function` auto-generates substitution rules from the tags — no manual type fixup is needed after cloning.

Section 2 — Using the generated artifacts

Listing 6: tutorials/macros/17_qmacro.das

```
options gen2
options no_aot

// Tutorial 17 - Quasi-quotation reference (qmacro variants).
//
// This tutorial demonstrates ALL qmacro variants and reification
// operators available in `daslib/templates_boost`.
//
// The companion module `qmacro_mod.das` defines a single
// [call_macro] named `build_demo` that uses every variant:
//
//   qmacro variants (10 + quote):
//     quote(expr)                     - plain AST capture, no reification
//     qmacro(expr)                    - expression with reification
```

(continues on next page)

(continued from previous page)

```
//     qmacro_type(type<T>)              - build TypeDeclPtr
//     qmacro_variable("n", type<T>)     - build VariablePtr
//     qmacro_expr(${ stmt })            - single statement
//     qmacro_block() { stmts }          - block of statements
//     qmacro_block_to_array() { stmts } - statements → array<ExpressionPtr>
//     qmacro_function("name") $(args) { body }       - full function
//     qmacro_template_class("Name", type<Template>)  - clone struct template
//     qmacro_method("Cls`meth", cls) $(self...) { }   - class method (*)
//     qmacro_template_function(@@fn)                   - clone template fn
//
//   qmacro_method is used here to generate a `get_tuple` method
//   on the cloned DemoIntFloat struct.
//
//   Reification operators:
//     $v(value)   - compile-time value → AST constant
//     $e(expr)    - splice existing ExpressionPtr
//     $i(name)    - string → identifier
//     $c(name)    - string → function call name
//     $f(name)    - string → field access name
//     $t(type)    - splice TypeDeclPtr in type position
//     $a(args)    - splice array<VariablePtr> as parameters
//     $b(body)    - splice array<ExpressionPtr> as block body
//
// See `qmacro_mod.das` for the implementation of each variant.
// The code below exercises the artefacts that the macro generates
// at compile time.

require qmacro_mod

[export]
def main() {
    // build_demo() triggers the call macro.  All functions and structs
    // are registered at compile time via add_function / add_structure.
    // The return value (a trivial constant) is discarded.
    build_demo()

    // 1. demo_run - generated by qmacro_function.
    //    Its body was assembled from qmacro, qmacro_expr, qmacro_block,
    //    qmacro_block_to_array, and reification operators
    //    $v, $e, $i, $c, $f, $a, $b.
    print("--- demo_run ---\n")
    demo_run("test")

    // 2. DemoIntFloat - generated by qmacro_template_class.
    //    Cloned from struct template DemoPair with TFirst=int, TSecond=float.
    print("\n--- DemoIntFloat (qmacro_template_class) ---\n")
    var pair = DemoIntFloat(first = 42, second = 3.14)
    // describe() was a template method on DemoPair - it was cloned
    // and type-substituted automatically by qmacro_template_class.
    pair.describe()

    let (a, b) = pair.get_tuple()
```

(continues on next page)

(continued from previous page)

```
    print("get_tuple() = ({a}, {b})\n")

    // 3. demo_add_int - generated by qmacro_template_function.
    //    Cloned from template demo_add, types fixed to int.
    print("\n--- demo_add_int (qmacro_template_function) ---\n")
    let sum = demo_add_int(10, 20)
    print("demo_add_int(10, 20) = {sum}\n")
}
```

`build_demo()` triggers the call macro. The generated artefacts:

1. **``demo_run(tag)``** — a function whose body was assembled from `qmacro`, `qmacro_expr`, `qmacro_block`, `qmacro_block_to_array`, and all eight reification operators.
2. **``DemoIntFloat``** — a struct cloned from the `DemoPair` template via `qmacro_template_class`. Type aliases `TFirst` and `TSecond` are registered on the cloned struct with `add_structure_alias`, so the compiler resolves them to `int` and `float`. The template's `describe()` method is also cloned and renamed to `DemoIntFloat`describe`. Additionally, `qmacro_method` adds a `get_tuple()` method that returns `(first, second)` as a tuple.
3. **``demo_add_int``** — a function cloned from the `demo_add` template. The template's `$t(t_value)` tag types are resolved to `int` automatically by `qmacro_template_function`.

Running the tutorial:

```
$ daslang tutorials/macros/17_qmacro.das
--- demo_run ---
hello from qmacro!
the literal is: true
called via $c
$i resolved tag = test
pair.$f first = 42
qmacro_block body:
  step 1
  step 2
  step 3
  extra A
  extra B
title = items

--- DemoIntFloat (qmacro_template_class) ---
first  = 42
second = 3.14
get_tuple() = (42, 3.14)

--- demo_add_int (qmacro_template_function) ---
demo_add_int(10, 20) = 30
```

See also:

Full source: `17_qmacro.das`, `qmacro_mod.das`

Previous tutorial: tutorial_macro_template_type_macro

Standard library: `daslib/templates_boost.das` — quasi-quotation infrastructure (all `qmacro_*` variants, `Template`, `apply_template`)

Language reference: *Macros* — full macro system documentation