Daslang Standard Library

Release 0.6.0

Boris Batkin

Feb 27, 2026

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Daslang standard library is a collection of modules that extend the language with commonly needed functionality — math, string manipulation, file I/O, serialization, regular expressions, AST manipulation, and more.

Some modules are implemented in C++ and are available by default (`builtin`, `math`, `strings`, `fio`, `rtti`, `ast`, `network`, `jobque`, `uriparser`). The rest are written in daslang itself and live in the `daslib/` directory; import them with `require daslib/<module_name>`.

1.1 Core runtime

- *builtin* — built-in runtime functions, operators, containers, smart pointers, and system infrastructure
- *math* — vector and scalar math, trigonometry, noise, matrix and quaternion operations
- *math_bits* — bit-level float/int/double reinterpretation helpers
- *math_boost* — angle conversions, projection matrices, plane math, color packing
- *fio* — file input/output, directory manipulation, process spawning
- *random* — LCG-based random number generators and distributions
- *network* — low-level TCP socket server
- *uriparser* — URI parsing, normalization, and file-name conversion (based on UriParser)
- *uriparser_boost* — URI component accessors and query-string helpers

1.2 Strings

- *strings* — core string manipulation: search, slice, conversion, builder, character groups
- *strings_boost* — split/join, replace, Levenshtein distance, formatting helpers
- *utf8_utils* — UTF-8 encoding and decoding utilities
- *temp_strings* — temporary string construction that avoids heap allocation
- *stringify* — `%stringify~` reader macro for embedding long strings at compile time
- *base64* — Base64 encoding and decoding

1.3 Regular expressions

- *regex* — regular expression matching and searching
- *regex_boost* — `%regex~` reader macro for compile-time regex construction

1.4 Reflection and AST

- *rtti* — runtime type information: type queries, module/function/variable iteration, compilation and simulation
- *ast* — compile-time AST access: expression/function/structure creation, visitor pattern, macro infrastructure
- *ast_boost* — AST convenience helpers: queries, annotation manipulation, expression generation, visitor utilities
- *ast_block_to_loop* — AST transform that converts block-based iteration to explicit loops (used by DECS)
- *ast_used* — collect all types used by a set of functions (for code generation)
- *quote* — AST quasiquotation for constructing syntax trees from inline code
- *type_traits* — compile-time type introspection and manipulation macros
- *typemacro_boost* — type macro and template structure support infrastructure
- *dynamic_cast_rtti* — runtime dynamic casting between class hierarchies

1.5 Functional and algorithms

- *functional* — higher-order functions: `filter`, `map`, `reduce`, `any`, `all`, `flatten`, `sorted`
- *algorithm* — binary search, topological sort, set operations on tables, array utilities
- *sort_boost* — custom comparator support for the built-in `sort`
- *linq* — LINQ-style query operations: select, where, order, group, join, aggregate, set operations
- *linq_boost* — macro support for LINQ query syntax

1.6 Data structures

- *flat_hash_table* — open-addressing flat hash table template
- *cuckoo_hash_table* — cuckoo hash table with O(1) worst-case lookup
- *bool_array* — packed boolean array (`BoolArray`) backed by `uint[]` storage
- *linked_list* — intrusive doubly-linked list data structure
- *soa* — Structure of Arrays transformation for cache-friendly data layout

1.7 Serialization and data

- *archive* — general-purpose binary serialization framework with `Serializer` / `Archive` pattern
- *json* — JSON parser and writer (`JsValue` variant, `read_json`, `write_json`)
- *json_boost* — automatic struct ↔ JSON conversion via reflection

1.8 Jobs and concurrency

- *jobque* — job queue primitives: channels, job status, lock boxes, atomics
- *jobque_boost* — `new_job` / `new_thread` helpers, channel iteration
- *apply_in_context* — cross-context function evaluation helpers
- *async_boost* — async/await coroutine macros using job queues

1.9 Macros and metaprogramming

- *templates* — `decltype` macro and `[template]` function annotation
- *templates_boost* — template application helpers: variable/type replacement, hygienic names
- *macro_boost* — miscellaneous macro manipulation utilities
- *contracts* — function argument contract annotations (`[expect_any_array]`, `[expect_any_table]`, etc.)
- *apply* — `apply` reflection pattern for struct field iteration
- *enum_trait* — compile-time enumeration trait queries
- *constexpr* — constant expression detection and substitution macro
- *bitfield_boost* — operator overloads for bitfield types
- *bitfield_trait* — reflection utilities for bitfield names and values
- *consume* — `consume` pattern for move-ownership argument passing
- *generic_return* — `[generic_return]` annotation for return type instantiation
- *remove_call_args* — AST transformation to remove specified call arguments
- *class_boost* — macros for extending class functionality and method binding

1.10 Control flow macros

- *defer* — `defer` and `defer_delete` — execute code at scope exit
- *if_not_null* — `if_not_null` safe-access macro
- *safe_addr* — `safe_addr` and `temp_ptr` — safe temporary pointer macros
- *static_let* — `static_let` — variables initialized once and persisted across calls
- *lpipe* — left-pipe operator macro (`<|`)
- *is_local* — `is_local_expr` / `is_scope_expr` AST query helpers

- *assert_once* — `assert_once` — assertion that fires only on first failure
- *unroll* — compile-time loop unrolling macro
- *instance_function* — `[instance_function]` annotation for struct method binding
- *array_boost* — `temp_array`, `array_view`, and `empty` helpers
- *heartbeat* — periodic heartbeat callback injection

1.11 Pattern matching

- *match* — `match` macro for structural pattern matching on variants and values

1.12 Entity component system

- *decs* — DECS (Daslang Entity Component System): archetypes, components, queries, staged updates
- *decs_boost* — DECS macro support for query syntax
- *decs_state* — DECS state machine support for entity lifecycle

1.13 OOP and interfaces

- *coroutines* — coroutine runner (`cr_run`, `cr_run_all`) and generator macros (`yield_from`)
- *interfaces* — `[interface]` and `[implements]` annotations for interface-based polymorphism
- *cpp_bind* — C++ class adapter binding code generator

1.14 Testing and debugging

- *faker* — random test data generator: strings, numbers, dates
- *fuzzer* — function fuzzing framework
- *coverage* — code coverage instrumentation and reporting
- *profiler* — instrumenting CPU profiler for function-level timing
- *profiler_boost* — profiler cross-context helpers and high-level macros
- *debug_eval* — runtime expression evaluation for interactive debugging
- *dap* — Debug Adapter Protocol (DAP) data structures for debugger integration

1.15 Code quality and tooling

- *lint* — static analysis pass for common code issues
- *lint_everything* — global lint pass applying paranoid diagnostics to all modules
- *validate_code* — AST validation annotations for custom code checks
- *refactor* — automated code refactoring transformations

1.16 Developer tools

- *das_source_formatter* — daslang source code formatter
- *das_source_formatter_fio* — file-based source code formatting
- *rst* — RST documentation generator used to produce this reference

CHAPTER

TWO

CORE

Core modules providing built-in operations, math, and random number generation.

2.1 Built-in runtime

The BUILTIN module contains core runtime functions available in all daslang programs without explicit `require`. It includes:

- Heap and memory management (`heap_bytes_allocated`, `heap_report`, `memory_report`)
- Debug output (`print`, `debug`, `stackwalk`)
- Panic and error handling (`panic`, `terminate`, `assert`)
- Pointer and memory operations (`intptr`, `malloc`, `free`)
- Profiling (`profile`)
- Type conversion (`string`)

All functions and symbols are in "builtin" module, use require to get access to it.

```
require builtin
```

Example:

```
[export]
    def main() {
        print("hello, world!\n")
        assert(1 + 1 == 2)
        let s = string(42)
        print("string(42) = {s}\n")
        let name = "daslang"
        print("welcome to {name}\n")
        var arr : array<int>
        arr |> push(10)
        arr |> push(20)
        print("length = {length(arr)}\n")
        print("arr[0] = {arr[0]}\n")
    }
    // output:
    // hello, world!
    // string(42) = 42
```

(continues on next page)

(continued from previous page)

```
    // welcome to daslang
    // length = 2
    // arr[0] = 10
```

2.1.1 Type aliases

`_builtin::bitfield print_flags`

This bitfield specifies how exactly values are to be printed

Fields

- **escapeString** (0x1) - if string is to be escaped
- **namesAndDimensions** (0x2) - names of the fields and dimensions of the arrays
- **typeQualifiers** (0x4) - type qualifiers for the specific types like double and uint64
- **refAddresses** (0x8) - addresses in hexadecimal of each reference value
- **singleLine** (0x10) - human readable vs single line
- **fixedPoint** (0x20) - always output fixed point precision for floating point values

2.1.2 Constants

`_builtin::DAS_MAX_FUNCTION_ARGUMENTS = 32`

Maximum number of arguments a function can accept, used to pre-allocate stack space for function call arguments.

`_builtin::INT_MIN = -2147483648`

Minimum representable value of a signed 32-bit integer (*int*), equal to -2147483648.

`_builtin::INT_MAX = 2147483647`

Maximum representable value of a signed 32-bit integer (*int*), equal to 2147483647.

`_builtin::UINT_MAX = 0xffffffff`

Maximum representable value of an unsigned 32-bit integer (*uint*), equal to 4294967295.

`_builtin::LONG_MIN = -9223372036854775808`

Minimum representable value of a signed 64-bit integer (*int64*).

`_builtin::LONG_MAX = 9223372036854775807`

Maximum representable value of a signed 64-bit integer (*int64*).

`_builtin::ULONG_MAX = 0xffffffffffffffff`

Maximum representable value of an unsigned 64-bit integer (*uint64*).

`FLT_MIN = 1.1754944e-38f`

Smallest positive non-zero normalized value of the *float* type; for the most negative value use *-FLT_MAX*.

`FLT_MAX = 3.4028235e+38f`

Maximum finite representable value of the *float* (32-bit floating-point) type.

`DBL_MIN = 2.2250738585072014e-308lf`

Smallest positive non-zero normalized value of the *double* type; for the most negative value use *-DBL_MAX*.

`DBL_MAX = 1.7976931348623157e+308lf`

Maximum finite representable value of the *double* (64-bit floating-point) type.

`_builtin::LOG_CRITICAL = 50000`

Log level constant for critical errors such as panics, fatal failures, and shutdown notifications.

`_builtin::LOG_ERROR = 40000`

Log level constant for recoverable error conditions that do not require immediate shutdown.

`_builtin::LOG_WARNING = 30000`

Log level constant for warnings about potential problems, API misuse, or non-fatal error conditions.

`_builtin::LOG_INFO = 20000`

Log level constant for general informational messages about normal program operation.

`_builtin::LOG_DEBUG = 10000`

Log level constant for debug-level diagnostic messages useful during development.

`_builtin::LOG_TRACE = 0`

Log level constant for the most verbose tracing and diagnostic output, typically used for detailed debugging.

`_builtin::VEC_SEP = ","`

Read-only string constant used as the separator between vector components when printing; defaults to *","*.

`_builtin::print_flags_debugger = bitfield`

Predefined set of print_flags configured to match the output formatting used by the *debug* function.

2.1.3 Handled structures

`_builtin::HashBuilder`

Helper structure to facilitate calculating hash values.

2.1.4 Function annotations

`_builtin::marker`

Attaches arbitrary key-value annotation arguments to a function, typically used by macros to tag functions with metadata.

`_builtin::generic`

Forces a function to be treated as generic regardless of its argument types, causing it to be instanced in each calling module.

`_builtin::_macro`

Marks a function to be executed during the macro compilation pass, similar to *[init]* but running at macro time.

`_builtin::macro_function`

Marks a function as part of the macro subsystem, excluding it from the final compiled context unless it is explicitly referenced.

`_builtin::hint`

Provides optimization hints to the compiler for the annotated function via annotation arguments.

`_builtin::jit`

Explicitly forces the annotated function to be compiled using the JIT compiler, overriding default compilation decisions.

`_builtin::no_jit`

Prevents JIT compilation for the annotated function, forcing it to run in interpreted mode.

`_builtin::nodiscard`

Enforces that the return value of the function must be used by the caller; discarding the result produces a compilation error.

`_builtin::deprecated`

Marks a function as deprecated, causing a compilation warning when referenced and excluding it from the final compiled context.

`_builtin::alias_cmres`

Declares that the function always aliases cmres (copy-or-move result), disabling cmres return optimizations for it.

`_builtin::never_alias_cmres`

Declares that the function never aliases cmres (copy-or-move result), disabling aliasing safety checks for the return value.

`_builtin::export`

Forces a function to be exported and retained in the final compiled context, even if it is not directly called.

`_builtin::pinvoke`

Marks a function as a platform invoke (pinvoke) entry, routing its calls through the pinvoke interop machinery.

`_builtin::no_lint`

Skips all lint-pass checks for the annotated function, suppressing any lint warnings or errors it would produce.

`_builtin::sideeffects`

Declares that the function has side effects, preventing the compiler from optimizing away or reordering its calls.

`_builtin::run`

Forces the function to be evaluated at compile time, ensuring its body executes during compilation rather than at runtime.

`_builtin::unsafe_operation`

Marks a function as an unsafe operation, requiring callers to wrap the call in an *unsafe* block.

`_builtin::unsafe_outside_of_for`

Marks a function as unsafe to call outside of a source-level *for* loop, enforcing iterator-context usage.

`_builtin::unsafe_when_not_clone_array`

Marks a function as unsafe to call outside of an array *clone* operation, restricting its usage context.

`_builtin::no_aot`

Prevents ahead-of-time (AOT) C++ code generation for the annotated function, keeping it interpreted only.

`_builtin::init`

Registers a function to be called automatically during context initialization, before any user code runs.

`_builtin::finalize`

Registers a function to be called automatically when the context is shut down, for cleanup and resource release.

`_builtin::hybrid`

Marks a function as a hybrid call target so that AOT generates indirect calls to it, allowing the function to be patched without recompiling dependent AOT code.

`_builtin::unsafe_deref`

Optimization annotation that removes null-pointer checks, bounds checks on array and string indexing, and similar safety validations.

`_builtin::skip_lock_check`

Optimization annotation that disables runtime lock-check validation for the annotated function.

`_builtin::unused_argument`

Suppresses unused-argument warnings or errors for specific function parameters, providing a workaround when strict code policies are enabled.

`_builtin::local_only`

Restricts a function to accept only local *make* expressions such as structure initializers and tuple constructors.

`_builtin::expect_any_vector`

Contract annotation restricting a function argument to accept only *das::vector* template types.

`_builtin::expect_dim`

Contract annotation requiring a function argument to be a fixed-size (statically dimensioned) array.

`_builtin::expect_ref`

Contract annotation requiring a function argument to be passed by reference.

`_builtin::type_function`

Marks a function as a type function, meaning it operates on types at compile time and does not generate runtime code.

`_builtin::builtin_array_sort`

Internal function annotation that provides the sorting implementation used by the built-in *sort* function.

2.1.5 Call macros

`_builtin::`**`make_function_unsafe`**

Propagates the *unsafe* requirement to the calling function, making any function that calls it also require an *unsafe* block.

`_builtin::`**`concept_assert`**

Compile-time assertion that reports the error at the call site of the asserted function rather than at the assert line itself.

`_builtin::`**`__builtin_table_set_insert`**

Internal function annotation that implements the low-level key insertion for set-style tables (tables with keys only).

`_builtin::`**`__builtin_table_key_exists`**

Internal function annotation that implements the low-level key presence check for the *key_exists* operation.

`_builtin::`**`static_assert`**

Compile-time assertion that produces a compilation error with an optional message when the condition is false.

`_builtin::`**`verify`**

Assertion that preserves the evaluated expression even when asserts are disabled, ensuring side effects are never optimized out.

`_builtin::`**`debug`**

Prints the human-readable representation of a value to the log and returns that same value, allowing inline debugging in expressions.

`_builtin::`**`assert`**

Runtime assertion that panics with an optional error message when the first argument evaluates to false; can be disabled globally.

`_builtin::`**`memzero`**

Fills a region of memory with zeros, used internally for default-initializing values.

`_builtin::`**`__builtin_table_find`**

Internal function annotation that implements the low-level table lookup for the *find* operation.

`_builtin::`**`invoke`**

Invokes a block, function pointer, or lambda, dispatching the call through the appropriate calling convention.

`_builtin::`**`__builtin_table_erase`**

Internal function annotation that implements the low-level table entry removal for the *erase* operation.

2.1.6 Reader macros

`_builtin::_esc`

Reader macro that returns the raw string content without processing escape sequences, e.g. *%_esc~nr~%_esc* yields the literal four characters `, `*n*, `, `*r*.

2.1.7 Typeinfo macros

`_builtin::rtti_classinfo`

Typeinfo macro that generates RTTI *TypeInfo* metadata required for class initialization and reflection.

2.1.8 Handled types

`_builtin::das_string`

Handled type wrapping *das::string* (typically *std::string*), providing heap-allocated mutable string storage.

`_builtin::clock`

Handled type wrapping *das::Time*, which encapsulates the C *time_t* value for calendar time representation.

2.1.9 Structure macros

`_builtin::comment`

No-op structure annotation that holds annotation arguments as metadata without affecting code generation.

`_builtin::no_default_initializer`

Prevents the compiler from generating a default initializer for the annotated structure.

`_builtin::macro_interface`

Marks a class hierarchy as a macro interface, preventing it and its descendants from being exported by default.

`_builtin::skip_field_lock_check`

Optimization annotation that disables runtime lock checks when accessing fields of the annotated structure.

`_builtin::cpp_layout`

Forces the structure to use C++ memory layout rules (alignment and padding) instead of native daslang layout.

`_builtin::safe_when_uninitialized`

Declares that the structure is safe to access before explicit initialization, suppressing uninitialized-use errors.

`_builtin::persistent`

Allocates the structure on the C++ heap (via *new*) instead of the daslang context heap, allowing it to outlive the context.

2.1.10 Containers

- *back (a: array<auto(TT)>#) : TT const&#*
- *back (a: array<auto(TT)>) : TT*
- *back (var a: array<auto(TT)>#) : TT&#*
- *back (var arr: auto(TT) ==const) : auto&*
- *back (arr: auto(TT) ==const) : auto*
- *back (var a: array<auto(TT)>) : TT&*
- *capacity (table: table<anything, anything>) : int*
- *capacity (array: array<anything>) : int*
- *clear (array: array<anything>)*
- *clear (var t: table<auto(KT), auto(VT)>) : auto*
- *copy_to_local (a: auto(TT)) : TT*
- *each (a: array<auto(TT)>#) : iterator<TT&#>*
- *each (str: string) : iterator<int>*
- *each (a: array<auto(TT)>) : iterator<TT&>*
- *each (a: auto(TT)[]) : iterator<TT&>*
- *each (lam: lambda<(var arg:auto(argT)):bool>) : iterator<argT>*
- *each (rng: range) : iterator<int>*
- *each (rng: urange64) : iterator<uint64>*
- *each (rng: range64) : iterator<int64>*
- *each (rng: urange) : iterator<uint>*
- *each_enum (tt: auto(TT)) : iterator<TT>*
- *each_ref (lam: lambda<(var arg:auto(argT)?):bool>) : iterator<argT&>*
- *emplace (var Arr: array<auto(numT)>; var value: numT&; at: int) : auto*
- *emplace (var Arr: array<auto(numT)>; var value: numT&) : auto*
- *emplace (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; var val: valT[]&) : auto*
- *emplace (var Tab: table<auto, auto>; key: auto; value: auto) : auto*
- *emplace (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; var val: valT&) : auto*
- *emplace (var Tab: table<auto(keyT), smart_ptr<auto(valT)>>; at: keyT|keyT#; var val: smart_ptr<valT>&) : auto*
- *emplace (var a: array<auto>; value: auto) : auto*
- *emplace (var Arr: array<auto(numT)>; var value: numT[]) : auto*
- *emplace (var Arr: array<auto(numT)[]>; var value: numT[]) : auto*
- *emplace_default (var tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#)*
- *emplace_new (var Arr: array<smart_ptr<auto(numT)>>; var value: smart_ptr<numT>) : auto*
- *emplace_new (var tab: table<auto(kT), smart_ptr<auto(vT)>>; key: kT; var value: smart_ptr<vT>) : auto*

- *empty (str: string) : bool*
- *empty (str: das_string) : bool*
- *empty (a: table<auto;auto>|table<auto;auto>#) : bool*
- *empty (a: array<auto>|array<auto>#) : bool*
- *empty (iterator: iterator) : bool*
- *erase (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : bool*
- *erase (var Arr: array<auto(numT)>; at: int) : auto*
- *erase (var Arr: array<auto(numT)>; at: int; count: int) : auto*
- *erase (var Tab: table<auto(keyT), auto(valT)>; at: string#) : bool*
- *erase_if (var arr: array<auto(TT)>; blk: block<(key:TT const):bool>|block<(var key:TT&):bool>) : auto*
- *find_index (arr: array<auto(TT)>|array<auto(TT)>#; key: TT) : auto*
- *find_index (var arr: iterator<auto(TT)>; key: TT) : auto*
- *find_index (arr: auto(TT)[]|auto(TT)[]#; key: TT) : auto*
- *find_index_if (var arr: iterator<auto(TT)>; blk: block<(key:TT):bool>) : auto*
- *find_index_if (arr: array<auto(TT)>|array<auto(TT)>#; blk: block<(key:TT):bool>) : auto*
- *find_index_if (arr: auto(TT)[]|auto(TT)[]#; blk: block<(key:TT):bool>) : auto*
- *get (Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(p:valT):void>) : auto*
- *get (Tab: table<auto(keyT), auto(valT)>#; at: keyT|keyT#; blk: block<(p:valT const&#):void>) : auto*
- *get (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(var p:valT&):void>) : auto*
- *get (var Tab: table<auto(keyT), auto(valT)>#; at: keyT|keyT#; blk: block<(var p:valT&#):void>) : auto*
- *get (Tab: table<auto(keyT), auto(valT)[]>#; at: keyT|keyT#; blk: block<(p:valT const[-2]&#):void>) : auto*
- *get (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; blk: block<(var p:valT[-2]&):void>) : auto*
- *get (Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; blk: block<(p:valT const[-2]&):void>) : auto*
- *get (Tab: table<auto(keyT), void>; at: keyT|keyT#; blk: block<(var p:void?):void>) : auto*
- *get (var Tab: table<auto(keyT), auto(valT)[]>#; at: keyT|keyT#; blk: block<(var p:valT[-2]&#):void>) : auto*
- *get_value (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#) : valT[-2]*
- *get_value (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT*
- *get_value (Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT*
- *get_value (var Tab: table<auto(keyT), smart_ptr<auto(valT)>>; at: keyT|keyT#) : smart_ptr<valT>*
- *get_with_default (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(var p:valT&):void>)*
- *has_value (var a: iterator<auto>; key: auto) : auto*
- *has_value (a: auto; key: auto) : auto*
- *insert (var Tab: table<auto(keyT), void>; at: keyT|keyT#) : auto*
- *insert (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT const# ==const) : auto*
- *insert (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; var val: valT[] ==const|valT[]# ==const) : auto*

- *insert (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; var val: valT ==const|valT# ==const) : auto*
- *insert (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT const[] ==const|valT const[]# ==const) : auto*
- *insert_clone (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; var val: valT ==const|valT# ==const) : auto*
- *insert_clone (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT const# ==const) : auto*
- *insert_clone (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; var val: valT[] ==const|valT[]# ==const) : auto*
- *insert_clone (var Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT const[] ==const|valT const[]# ==const) : auto*
- *insert_default (var tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#; value: valT ==const|valT const# ==const)*
- *insert_default (var tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#)*
- *insert_default (var tab: table<auto(TT), auto(QQ)>; key: TT|TT#; var value: QQ ==const|QQ# ==const)*
- *key_exists (Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; at: keyT|keyT#) : bool*
- *key_exists (Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; at: string#) : bool*
- *keys (var a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)># ==const) : iterator<keyT>*
- *keys (a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)> const# ==const) : iterator<keyT>*
- *length (a: auto|auto#) : int*
- *length (array: array<anything>) : int*
- *length (table: table<anything, anything>) : int*
- *lock (a: array<auto(TT)> ==const|array<auto(TT)> const# ==const; blk: block<(x:array<TT>#):auto>) : auto*
- *lock (var a: array<auto(TT)> ==const|array<auto(TT)># ==const; blk: block<(var x:array<TT>#):auto>) : auto*
- *lock (Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; blk: block<(t:table<keyT, valT>#):void>) : auto*
- *lock_forever (var Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#) : table<keyT, valT>#*
- *modify (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(p:valT):valT>)*
- *move_to_local (var a: auto(TT)&) : TT*
- *move_to_ref (var a: auto&; var b: auto) : auto*
- *next (var it: iterator<auto(TT)>; var value: TT&) : bool*
- *nothing (var it: iterator<auto(TT)>) : iterator<TT>*
- *pop (var Arr: array<auto(numT)>) : auto*
- *push (var Arr: array<auto(numT)[]>; varr: numT const[] ==const) : auto*
- *push (var Arr: array<auto(numT)>; varr: numT[]) : auto*
- *push (var Arr: array<auto(numT)[]>; var varr: numT[] ==const) : auto*

- *push (var Arr: array<auto(numT)>; varr: array<numT>) : auto*
- *push (var Arr: array<auto(numT)>; value: numT ==const) : auto*
- *push (var Arr: array<auto(numT)>; var value: numT ==const; at: int) : auto*
- *push (var Arr: array<auto(numT)>; var value: numT ==const) : auto*
- *push (var Arr: array<auto(numT)>; var varr: array<numT>) : auto*
- *push (var Arr: array<auto(numT)>; value: numT ==const; at: int) : auto*
- *push_clone (var A: auto(CT); b: auto(TT)|auto(TT)#) : auto*
- *push_clone (var Arr: array<auto(numT)[]>; var varr: numT[]) : auto*
- *push_clone (var Arr: array<auto(numT)>; varr: numT const[] ==const) : auto*
- *push_clone (var Arr: array<auto(numT)>; var value: numT ==const|numT# ==const) : auto*
- *push_clone (var Arr: array<auto(numT)>; var varr: numT[] ==const) : auto*
- *push_clone (var Arr: array<auto(numT)>; var value: numT ==const|numT# ==const; at: int) : auto*
- *push_clone (var Arr: array<auto(numT)>; value: numT ==const|numT const# ==const; at: int) : auto*
- *push_clone (var Arr: array<auto(numT)>; value: numT ==const|numT const# ==const) : auto*
- *push_clone (var Arr: array<auto(numT)[]>; varr: numT const[] ==const) : auto*
- *remove_value (var arr: array<auto(TT)>|array<auto(TT)>#; key: TT) : bool*
- *reserve (var Tab: table<auto(keyT), auto>; newSize: int) : auto*
- *reserve (var Arr: array<auto(numT)>; newSize: int) : auto*
- *resize (var Arr: array<auto(numT)>; newSize: int) : auto*
- *resize_and_init (var Arr: array<auto(numT)>; newSize: int; initValue: numT) : auto*
- *resize_and_init (var Arr: array<auto(numT)>; newSize: int) : auto*
- *resize_no_init (var Arr: array<auto(numT)>; newSize: int) : auto*
- *sort (var a: array<auto(TT)>|array<auto(TT)>#) : auto*
- *sort (var a: array<auto(TT)>|array<auto(TT)>#; cmp: block<(x:TT;y:TT):bool>) : auto*
- *sort (var a: auto(TT)[]|auto(TT)[]#) : auto*
- *sort (var a: auto(TT)[]|auto(TT)[]#; cmp: block<(x:TT;y:TT):bool>) : auto*
- *subarray (a: auto(TT)[]; r: urange) : auto*
- *subarray (a: array<auto(TT)>; r: urange) : auto*
- *subarray (a: array<auto(TT)>; r: range) : auto*
- *subarray (var a: array<auto(TT)>; r: range) : auto*
- *subarray (a: auto(TT)[]; r: range) : auto*
- *to_array (a: auto(TT)[]) : array<TT>*
- *to_array (var it: iterator<auto(TT)>) : array<TT>*
- *to_array_move (var a: auto(TT) ==const) : array<TT>*
- *to_array_move (var a: auto(TT)[]) : array<TT>*
- *to_array_move (a: auto(TT) ==const) : array<TT>*

- *to_table (a: tuple<auto(keyT);auto(valT)>[]) : table<keyT, valT>*
- *to_table (a: auto(keyT)[]) : table<keyT, void>*
- *to_table_move (var a: tuple<auto(keyT);auto(valT)>[]) : table<keyT, valT>*
- *to_table_move (var a: tuple<auto(keyT);auto(valT)>) : table<keyT, valT>*
- *to_table_move (var a: array<tuple<auto(keyT);auto(valT)>>) : table<keyT, valT>*
- *to_table_move (a: auto(keyT)[]) : table<keyT, void>*
- *to_table_move (a: array<auto(keyT)>) : table<keyT, void>*
- *to_table_move (a: auto(keyT)) : table<keyT, void>*
- *values (var a: table<auto(keyT);auto(valT)[]> ==const|table<auto(keyT);auto(valT)[]># ==const) : iterator<valT[-2]&>*
- *values (a: table<auto(keyT);auto(valT)[]> ==const|table<auto(keyT);auto(valT)[]> const# ==const) : iterator<valT const[-2]&>*
- *values (a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)> const# ==const) : iterator<valT const&>*
- *values (a: table<auto(keyT);void> ==const|table<auto(keyT);void> const# ==const) : auto*
- *values (var a: table<auto(keyT);void> ==const|table<auto(keyT);void># ==const) : auto*
- *values (var a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)># ==const) : iterator<valT&>*

back

`_builtin::back(a: array<auto(TT)>#) : TT const&#()`

Accesses and returns a const temporary reference to the last element of the temporary dynamic array *a*.

Arguments

- **a** : array<auto(TT)>#!

`_builtin::back(a: array<auto(TT)>) : TT()`

`_builtin::back(a: array<auto(TT)>#) : TT&#()`

`_builtin::back(arr: auto(TT) ==const) : auto&()`

`_builtin::back(arr: auto(TT) ==const) : auto()`

`_builtin::back(a: array<auto(TT)>) : TT&()`

capacity

`_builtin::capacity(table: table<anything, anything>) : int()`

Returns the current capacity of the *table* — the number of key-value pairs it can hold before triggering a reallocation.

Arguments

- **table** : table implicit

`_builtin::capacity(array: array<anything>) : int()`

---

clear

`_builtin::clear(array: array<anything>)`

Removes all elements from the dynamic *array*, leaving it empty with a size of 0.

Arguments

- **array** : array implicit

`_builtin::clear(t: table<auto(KT), auto(VT)>) : auto()`

`_builtin::copy_to_local(a: auto(TT)) : TT()`

Copies the value *a* and returns it as a new local value on the stack, which can be used to work around aliasing issues where a reference might be invalidated.

Arguments

- **a** : auto(TT)

each

`_builtin::each(a: array<auto(TT)>#) : iterator<TT&#>()`

Creates an iterator that yields temporary references to each element of the temporary dynamic array *a*.

Arguments

- **a** : array<auto(TT)>#

`_builtin::each(str: string) : iterator<int>()`

`_builtin::each(a: array<auto(TT)>) : iterator<TT&>()`

`_builtin::each(a: auto(TT)[]) : iterator<TT&>()`

`_builtin::each(lam: lambda<(var arg:auto(argT)):bool>) : iterator<argT>()`

`_builtin::each(rng: range) : iterator<int>()`

`_builtin::each(rng: urange64) : iterator<uint64>()`

`_builtin::each(rng: range64) : iterator<int64>()`

`_builtin::each(rng: urange) : iterator<uint>()`

`_builtin::each_enum(tt: auto(TT)) : iterator<TT>()`

Warning: This function is deprecated.

Creates an iterator that yields every value of the enumeration type inferred from *tt*, allowing iteration over all members of an enum.

Arguments

- **tt** : auto(TT)

`_builtin::each_ref(lam: lambda<(var arg:auto(argT)?):bool>) : iterator<argT&>()`

Wraps a lambda *lam* — which receives a mutable pointer argument and returns a bool indicating whether to continue — into an iterator that yields references to each value rather than copies.

Arguments

- **lam** : lambda<(arg:auto(argT)?):bool>

emplace

`_builtin::emplace(Arr: array<auto(numT)>; value: numT&; at: int) : auto()`

Moves *value* into the dynamic array *Arr* at index *at* using move semantics, shifting subsequent elements forward.

Arguments

- **Arr** : array<auto(numT)>
- **value** : numT&
- **at** : int

`_builtin::emplace(Arr: array<auto(numT)>; value: numT&) : auto()`

`_builtin::emplace(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT[]&) : auto()`

`_builtin::emplace(Tab: table<auto, auto>; key: auto; value: auto) : auto()`

`_builtin::emplace(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT&) : auto()`

`_builtin::emplace(Tab: table<auto(keyT), smart_ptr<auto(valT)>>; at: keyT|keyT#; val: smart_ptr<valT>&)`

`_builtin::emplace(a: array<auto>; value: auto) : auto()`

`_builtin::emplace(Arr: array<auto(numT)>; value: numT[]) : auto()`

`_builtin::emplace(Arr: array<auto(numT)[]>; value: numT[]) : auto()`

`_builtin::emplace_default(tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#)`

Constructs a new default-initialized element in the table *tab* at the given *key*, only if that key does not already exist.

Arguments

- **tab** : table<auto(keyT);auto(valT)>
- **key** : option<keyT| keyT#>

emplace_new

`_builtin::emplace_new(Arr: array<smart_ptr<auto(numT)>>; value: smart_ptr<numT>) : auto()`

Moves a smart pointer *value* into the end of the array *Arr*, constructing the entry in-place and returning a reference to it.

Arguments

- **Arr** : array<smart_ptr<auto(numT)>>
- **value** : smart_ptr<numT>

`_builtin::emplace_new(tab: table<auto(kT), smart_ptr<auto(vT)>>; key: kT; value: smart_ptr<vT>) : auto(`

---

empty

`_builtin::empty(str: string) : bool()`

Checks whether the *das_string* is empty (has zero length) and returns *true* if so.

Arguments

- **str** : string implicit

`_builtin::empty(str: das_string) : bool()`

`_builtin::empty(a: table<auto;auto>|table<auto;auto>#) : bool()`

`_builtin::empty(a: array<auto>|array<auto>#) : bool()`

`_builtin::empty(iterator: iterator) : bool()`

---

erase

`_builtin::erase(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : bool()`

Removes the entry with key *at* from the table *Tab*, returning *true* if the key was found and erased.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>
- **at** : option<keyT| keyT#>

_builtin::**erase(Arr: array<auto(numT)>; at: int) : auto()**

_builtin::**erase(Arr: array<auto(numT)>; at: int; count: int) : auto()**

_builtin::**erase(Tab: table<auto(keyT), auto(valT)>; at: string#) : bool()**

_builtin::**erase_if(arr: array<auto(TT)>; blk: block<(key:TT const):bool>|block<(var key:TT&):bool>) : au**

Iterates over the array *arr* and removes all elements for which the block *blk* returns *true*.

Arguments

- **arr** : array<auto(TT)>
- **blk** : option<block<(key:TT):bool>| block<(key:TT&):bool>>

find_index

_builtin::**find_index(arr: array<auto(TT)>|array<auto(TT)>#; key: TT) : auto()**

Searches the dynamic array *arr* for the first occurrence of *key* and returns its index, or -1 if not found.

Arguments

- **arr** : option<array<auto(TT)>| array<auto(TT)>#>
- **key** : TT

_builtin::**find_index(arr: iterator<auto(TT)>; key: TT) : auto()**

_builtin::**find_index(arr: auto(TT)[]|auto(TT)[]#; key: TT) : auto()**

---

find_index_if

_builtin::**find_index_if(arr: iterator<auto(TT)>; blk: block<(key:TT):bool>) : auto()**

Consumes elements from the iterator *arr* and returns the index of the first element for which the block *blk* returns *true*, or -1 if no match is found.

Arguments

- **arr** : iterator<auto(TT)>
- **blk** : block<(key:TT):bool>

_builtin::**find_index_if(arr: array<auto(TT)>|array<auto(TT)>#; blk: block<(key:TT):bool>) : auto()**

_builtin::**find_index_if(arr: auto(TT)[]|auto(TT)[]#; blk: block<(key:TT):bool>) : auto()**

---

get

`_builtin::get(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(p:valT):void>) : auto()`

Looks up *at* in the table *Tab* and, if found, invokes *blk* with a reference to the value; returns *true* if the key existed, *false* otherwise.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>!
- **at** : option<keyT| keyT#>
- **blk** : block<(p:valT&):void>

`_builtin::get(Tab: table<auto(keyT), auto(valT)>#; at: keyT|keyT#; blk: block<(p:valT const&#):void>) :`

`_builtin::get(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(var p:valT&):void>) : aut`

`_builtin::get(Tab: table<auto(keyT), auto(valT)>#; at: keyT|keyT#; blk: block<(var p:valT&#):void>) : a`

`_builtin::get(Tab: table<auto(keyT), auto(valT)[]>#; at: keyT|keyT#; blk: block<(p:valT const[-2]&#):vo`

`_builtin::get(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; blk: block<(var p:valT[-2]&):void>)`

`_builtin::get(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; blk: block<(p:valT const[-2]&):void`

`_builtin::get(Tab: table<auto(keyT), void>; at: keyT|keyT#; blk: block<(var p:void?):void>) : auto()`

`_builtin::get(Tab: table<auto(keyT), auto(valT)[]>#; at: keyT|keyT#; blk: block<(var p:valT[-2]&#):void`

---

get_value

`_builtin::get_value(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#) : valT[-2]()`

Retrieves the fixed-size array value associated with key *at* from the mutable table *Tab*.

Arguments

- **Tab** : table<auto(keyT);auto(valT)[-1]>
- **at** : option<keyT| keyT#>

`_builtin::get_value(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT()`

`_builtin::get_value(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT()`

`_builtin::get_value(Tab: table<auto(keyT), smart_ptr<auto(valT)>>; at: keyT|keyT#) : smart_ptr<valT>()`

`_builtin::get_with_default`(*Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(var p:valT&):void>*)

Looks up key *at* in the table *Tab*, inserting a default-initialized entry if the key is absent, then invokes *blk* with a mutable reference to the value.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>!

- **at** : option<keyT| keyT#>
- **blk** : block<(p:valT&):void>

has_value

`_builtin::has_value(a: iterator<auto>; key: auto) : auto()`

Consumes elements from the iterator *a* and returns *true* if any element equals *key*.

Arguments

- **a** : iterator<auto>
- **key** : auto

`_builtin::has_value(a: auto; key: auto) : auto()`

---

insert

`_builtin::insert(Tab: table<auto(keyT), void>; at: keyT|keyT#) : auto()`

Inserts the key *at* into the set-style table *Tab* (a table with *void* values), effectively adding to a set.

Arguments

- **Tab** : table<auto(keyT);void>
- **at** : option<keyT| keyT#>

`_builtin::insert(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT const# ==co`

`_builtin::insert(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT[] ==const|valT[]# ==co`

`_builtin::insert(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT# ==const) :`

`_builtin::insert(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT const[] ==const|valT c`

---

insert_clone

`_builtin::insert_clone(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT# ==co`

Inserts or updates an entry in the table *Tab* at key *at* by cloning the mutable value *val* into the table.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>
- **at** : option<keyT| keyT#>
- **val** : option<valT!| valT#!>

`_builtin::insert_clone(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; val: valT ==const|valT const`

`_builtin::insert_clone(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT[] ==const|valT[]`

`_builtin::insert_clone(Tab: table<auto(keyT), auto(valT)[]>; at: keyT|keyT#; val: valT const[] ==const|`

---

insert_default

`_builtin::insert_default`(*tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#; value: valT ==const|valT const# ==const*)

Inserts key *key* with the given const *value* into table *tab* only if the key does not already exist; existing entries are left unchanged.

Arguments

- **tab** : table<auto(keyT);auto(valT)>
- **key** : option<keyT| keyT#>
- **value** : option<valT!| valT#!>

`_builtin::insert_default`(*tab: table<auto(keyT), auto(valT)>; key: keyT|keyT#*)

`_builtin::insert_default`(*tab: table<auto(TT), auto(QQ)>; key: TT|TT#; value: QQ ==const|QQ# ==const*)

---

key_exists

`_builtin::key_exists(Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; at: keyT|keyT#) :`

Checks whether the key *at* exists in the table *Tab* and returns *true* if found.

Arguments

- **Tab** : option<table<auto(keyT);auto(valT)>| table<auto(keyT);auto(valT)>#>
- **at** : option<keyT| keyT#>

`_builtin::key_exists(Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; at: string#) : bo`

---

keys

`_builtin::keys(a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)># ==const) : iterato`

Creates an iterator over all keys of the mutable table *a*, allowing enumeration of the table's key set.

Arguments

- **a** : option<table<auto(keyT);auto(valT)>!| table<auto(keyT);auto(valT)>#!>

`_builtin::keys(a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)> const# ==const) : i`

---

length

`_builtin::length(a: auto|auto#) : int()`

Returns the number of elements currently stored in a table or dynamic array *a*.

Arguments

- **a** : option<auto| auto#>

`_builtin::length(array: array<anything>) : int()`

`_builtin::length(table: table<anything, anything>) : int()`

---

lock

`_builtin::lock(a: array<auto(TT)> ==const|array<auto(TT)> const# ==const; blk: block<(x:array<TT>#):aut`

Locks a constant array for the duration of *blk*, preventing structural modifications while providing read-only access through a temporary reference.

Arguments

- **a** : option<array<auto(TT)>!| array<auto(TT)>#!>
- **blk** : block<(x:array<TT>#):auto>

`_builtin::lock(a: array<auto(TT)> ==const|array<auto(TT)># ==const; blk: block<(var x:array<TT>#):auto>`

`_builtin::lock(Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#; blk: block<(t:table<key`

`_builtin::lock_forever(Tab: table<auto(keyT);auto(valT)>|table<auto(keyT);auto(valT)>#) : table<keyT, v`

Permanently locks a table, preventing any future insertions, deletions, or structural modifications, and returns a temporary reference to it.

Arguments

- **Tab** : option<table<auto(keyT);auto(valT)>| table<auto(keyT);auto(valT)>#>

`_builtin::modify`(*Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#; blk: block<(p:valT):valT>*)

Looks up *at* in *Tab* and, if found, invokes *blk* with the current value, replacing it with the value returned by the block.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>!
- **at** : option<keyT| keyT#>
- **blk** : block<(p:valT&):valT>

`_builtin::move_to_local(a: auto(TT)&) : TT()`

Moves the value referenced by *a* onto the stack as a local copy and returns it, clearing the original; useful for resolving aliasing issues.

Arguments

- **a** : auto(TT)&

`_builtin::move_to_ref(a: auto&; b: auto) : auto()`

Moves *b* into the reference *a*; if *b* is a value type rather than a reference, it is copied instead of moved.

Arguments

- **a** : auto&
- **b** : auto

`_builtin::next(it: iterator<auto(TT)>; value: TT&) : bool()`

Advances the iterator *it* and stores the next element in *value*, returning true if an element was retrieved or false if the iterator is exhausted or null.

Arguments

- **it** : iterator<auto(TT)>
- **value** : TT&

`_builtin::nothing(it: iterator<auto(TT)>) : iterator<TT>()`

Produces an empty iterator of the same element type as *it* that yields no elements.

Arguments

- **it** : iterator<auto(TT)>

`_builtin::pop(Arr: array<auto(numT)>) : auto()`

Removes and discards the last element of *Arr*, reducing its length by one.

Arguments

- **Arr** : array<auto(numT)>

push

`_builtin::push(Arr: array<auto(numT)[]>; varr: numT const[] ==const) : auto()`

Appends a constant fixed-size array element *varr* to the end of *Arr*, an array of fixed-size arrays.

Arguments

- **Arr** : array<auto(numT)[-1]>
- **varr** : numT[-1]!

`_builtin::push(Arr: array<auto(numT)>; varr: numT[]) : auto()`

`_builtin::push(Arr: array<auto(numT)[]>; varr: numT[] ==const) : auto()`

`_builtin::push(Arr: array<auto(numT)>; varr: array<numT>) : auto()`

`_builtin::push(Arr: array<auto(numT)>; value: numT ==const) : auto()`

`_builtin::push(Arr: array<auto(numT)>; value: numT ==const; at: int) : auto()`

`_builtin::push(Arr: array<auto(numT)>; value: numT ==const) : auto()`

`_builtin::push(Arr: array<auto(numT)>; varr: array<numT>) : auto()`

`_builtin::push(Arr: array<auto(numT)>; value: numT ==const; at: int) : auto()`

push_clone

`_builtin::push_clone(A: auto(CT); b: auto(TT)|auto(TT)#) : auto()`

Clones and appends element *b* to the container *A*, using deep copy semantics rather than move or shallow copy.

Arguments

- **A** : auto(CT)
- **b** : option<auto(TT)| auto(TT)#>

`_builtin::push_clone(Arr: array<auto(numT)[]>; varr: numT[]) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; varr: numT const[] ==const) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; value: numT ==const|numT# ==const) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; varr: numT[] ==const) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; value: numT ==const|numT# ==const; at: int) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; value: numT ==const|numT const# ==const; at: int) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)>; value: numT ==const|numT const# ==const) : auto()`

`_builtin::push_clone(Arr: array<auto(numT)[]>; varr: numT const[] ==const) : auto()`

`_builtin::remove_value(arr: array<auto(TT)>|array<auto(TT)>#; key: TT) : bool()`

Searches *arr* for the first element equal to *key* and removes it, returning true if an element was found and removed or false otherwise.

Arguments

- **arr** : option<array<auto(TT)>| array<auto(TT)>#>
- **key** : TT

reserve

`_builtin::reserve(Tab: table<auto(keyT), auto>; newSize: int) : auto()`

Pre-allocates memory in *Tab* to hold at least *newSize* entries without rehashing, improving performance of subsequent insertions.

Arguments

- **Tab** : table<auto(keyT);auto>
- **newSize** : int

`_builtin::reserve(Arr: array<auto(numT)>; newSize: int) : auto()`

`_builtin::resize(Arr: array<auto(numT)>; newSize: int) : auto()`

Resizes dynamic array *Arr* to *newSize* elements; new elements beyond the previous length are zero-initialized, and excess elements are removed.

Arguments

- **Arr** : array<auto(numT)>
- **newSize** : int

resize_and_init

`_builtin::resize_and_init(Arr: array<auto(numT)>; newSize: int; initValue: numT) : auto()`

Resizes dynamic array *Arr* to *newSize* elements, initializing any newly added elements with the provided *initValue*.

Arguments

- **Arr** : array<auto(numT)>
- **newSize** : int
- **initValue** : numT

`_builtin::resize_and_init(Arr: array<auto(numT)>; newSize: int) : auto()`

`_builtin::resize_no_init(Arr: array<auto(numT)>; newSize: int) : auto()`

Resizes dynamic array *Arr* to *newSize* elements without initializing newly added entries, leaving their memory contents undefined.

Arguments

- **Arr** : array<auto(numT)>
- **newSize** : int

sort

`_builtin::sort(a: array<auto(TT)>|array<auto(TT)>#) : auto()`

Sorts a dynamic array in place in ascending order using the default comparison for its element type.

Arguments

- **a** : option<array<auto(TT)>| array<auto(TT)>#>

`_builtin::sort(a: array<auto(TT)>|array<auto(TT)>#; cmp: block<(x:TT;y:TT):bool>) : auto()`

`_builtin::sort(a: auto(TT)[]|auto(TT)[]#) : auto()`

`_builtin::sort(a: auto(TT)[]|auto(TT)[]#; cmp: block<(x:TT;y:TT):bool>) : auto()`

---

subarray

`_builtin::subarray(a: auto(TT)[]; r: urange) : auto()`

Creates and returns a new dynamic array containing a copy of elements from fixed-size array *a* within the unsigned range *r*.

Arguments

- **a** : auto(TT)[-1]
- **r** : urange

`_builtin::subarray(a: array<auto(TT)>; r: urange) : auto()`

`_builtin::subarray(a: array<auto(TT)>; r: range) : auto()`

`_builtin::subarray(a: array<auto(TT)>; r: range) : auto()`

`_builtin::subarray(a: auto(TT)[]; r: range) : auto()`

---

to_array

`_builtin::to_array(a: auto(TT)[]) : array<TT>()`

Converts a fixed-size array *a* into a new dynamic array by cloning each element.

Arguments

- **a** : auto(TT)[-1]

`_builtin::to_array(it: iterator<auto(TT)>) : array<TT>()`

---

to_array_move

`_builtin::to_array_move(a: auto(TT) ==const) : array<TT>()`

Converts a mutable container *a* into a new dynamic array, moving elements when possible instead of cloning.

Arguments

- **a** : auto(TT)!

`_builtin::to_array_move(a: auto(TT)[]) : array<TT>()`

`_builtin::to_array_move(a: auto(TT) ==const) : array<TT>()`

---

to_table

`_builtin::to_table(a: tuple<auto(keyT);auto(valT)>[]) : table<keyT, valT>()`

Converts a fixed-size array of key-value tuples *a* into a *table<keyT, valT>* by cloning each key and value.

Arguments

- **a** : tuple<auto(keyT);auto(valT)>[-1]

`_builtin::to_table(a: auto(keyT)[]) : table<keyT, void>()`

to_table_move

`_builtin::to_table_move(a: tuple<auto(keyT);auto(valT)>[]) : table<keyT, valT>()`

Converts a mutable fixed-size array of key-value tuples *a* into a *table<keyT, valT>*, moving elements when possible.

Arguments

- **a** : tuple<auto(keyT);auto(valT)>[-1]

`_builtin::to_table_move(a: tuple<auto(keyT);auto(valT)>) : table<keyT, valT>()`

`_builtin::to_table_move(a: array<tuple<auto(keyT);auto(valT)>>) : table<keyT, valT>()`

`_builtin::to_table_move(a: auto(keyT)[]) : table<keyT, void>()`

`_builtin::to_table_move(a: array<auto(keyT)>) : table<keyT, void>()`

`_builtin::to_table_move(a: auto(keyT)) : table<keyT, void>()`

values

`_builtin::values(a: table<auto(keyT);auto(valT)[]> ==const|table<auto(keyT);auto(valT)[]># ==const) : i`

Returns a mutable iterator over all fixed-size array values in a mutable *table<keyT, valT[]>*, yielding each array by reference.

Arguments

- **a** : option<table<auto(keyT);auto(valT)[-1]>!| table<auto(keyT);auto(valT)[-1]>#!>

`_builtin::values(a: table<auto(keyT);auto(valT)[]> ==const|table<auto(keyT);auto(valT)[]> const# ==cons`

`_builtin::values(a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)> const# ==const) :`

`_builtin::values(a: table<auto(keyT);void> ==const|table<auto(keyT);void> const# ==const) : auto()`

`_builtin::values(a: table<auto(keyT);void> ==const|table<auto(keyT);void># ==const) : auto()`

`_builtin::values(a: table<auto(keyT);auto(valT)> ==const|table<auto(keyT);auto(valT)># ==const) : itera`

2.1.11 das::string manipulation

- *peek (src: das_string; block: block<(string#):void>)*

`_builtin::`**`peek`**`(`*src: das_string; block: block<(string#):void>*`)`

Provides zero-copy read access to the contents of a *das_string* by invoking *block* with a temporary string reference, avoiding allocation.

Arguments

- **src** : *das_string* implicit
- **block** : block<(string#):void> implicit

2.1.12 Heap reporting

- *heap_allocation_count () : uint64*
- *heap_allocation_stats () : urange64*
- *heap_bytes_allocated () : uint64*
- *heap_collect (string_heap: bool = true; validate: bool = false)*
- *heap_depth () : int*
- *heap_report ()*
- *memory_report (errorsOnly: bool)*
- *string_heap_allocation_count () : uint64*
- *string_heap_allocation_stats () : urange64*
- *string_heap_bytes_allocated () : uint64*
- *string_heap_depth () : int*
- *string_heap_report ()*

`_builtin::`**`heap_allocation_count() : uint64`**`()`

Returns the total number of heap allocations performed by the current context since it was created.

`_builtin::`**`heap_allocation_stats() : urange64`**`()`

Returns heap allocation statistics as a *urange64*, where the *x* component is total bytes allocated and the *y* component is total bytes freed.

`_builtin::`**`heap_bytes_allocated() : uint64`**`()`

Returns the number of bytes currently in use on the heap (allocated minus freed), not counting reserved but unused memory.

`_builtin::`**`heap_collect`**`(`*string_heap: bool = true; validate: bool = false*`)`

Warning: This is unsafe operation.

Triggers garbage collection on the context heap; when *string_heap* is *true* the string heap is also collected, and when *validate* is *true* additional validation checks are performed.

Arguments

- **string_heap** : bool
- **validate** : bool

`_builtin::heap_depth() : int()`

Returns the number of generations (depth of the allocation chain) in the context's regular heap.

`_builtin::heap_report()`

Prints a diagnostic report of current heap usage and allocation statistics to the output log.

`_builtin::memory_report(errorsOnly: bool)`

Prints a report of memory allocations for the current context; when *errorsOnly* is true, only GC-related errors are included.

Arguments

- **errorsOnly** : bool

`_builtin::string_heap_allocation_count() : uint64()`

Returns the total number of individual string allocations performed on the current context's string heap.

`_builtin::string_heap_allocation_stats() : urange64()`

Returns string heap allocation statistics as a *urange64* where *x* is total bytes allocated and *y* is total bytes deleted.

`_builtin::string_heap_bytes_allocated() : uint64()`

Returns the total number of bytes currently allocated in the current context's string heap.

`_builtin::string_heap_depth() : int()`

Returns the number of generational layers (depth) in the current context's string heap.

`_builtin::string_heap_report()`

Prints a detailed report of string heap usage, including allocation counts and byte statistics, to the log output.

2.1.13 GC0 infrastructure

- *gc0_reset ()*
- *gc0_restore_ptr (name: string) : void?*
- *gc0_restore_smart_ptr (name: string) : smart_ptr<void>*
- *gc0_save_ptr (name: string; data: void?)*
- *gc0_save_smart_ptr (name: string; data: smart_ptr<void>)*

`_builtin::gc0_reset()`

Clears the entire gc0 storage, invalidating all previously saved pointers and smart pointers stored within it.

`_builtin::gc0_restore_ptr(name: string) : void?()`

Retrieves a raw pointer previously saved in gc0 storage under the specified *name*, returning *null* if not found.

Arguments

- **name** : string implicit

`_builtin::gc0_restore_smart_ptr(name: string) : smart_ptr<void>()`

Retrieves a *smart_ptr<void>* previously saved in gc0 storage under the specified *name*.

Arguments

- **name** : string implicit

`_builtin::gc0_save_ptr(`*name: string; data: void?*`)`

Stores a raw pointer *data* into gc0 storage under the specified *name*, allowing it to be retrieved later with *gc0_restore_ptr*.

Arguments

- **name** : string implicit
- **data** : void? implicit

`_builtin::gc0_save_smart_ptr(`*name: string; data: smart_ptr<void>*`)`

Stores a *smart_ptr<void>* *data* into gc0 storage under the specified *name*, allowing it to be retrieved later with *gc0_restore_smart_ptr*.

Arguments

- **name** : string implicit
- **data** : smart_ptr<void> implicit

2.1.14 Smart ptr infrastructure

- *add_ptr_ref (src: auto(TT)?) : smart_ptr<TT>*
- *add_ptr_ref (src: smart_ptr<auto(TT)>) : smart_ptr<TT>*
- *get_const_ptr (src: smart_ptr<auto(TT)>) : TT?*
- *get_ptr (var src: smart_ptr<auto(TT)> ==const) : TT?*
- *get_ptr (src: smart_ptr<auto(TT)> ==const) : TT?*
- *move (dest: smart_ptr<void>&; src: void?)*
- *move (dest: smart_ptr<void>&; src: smart_ptr<void>&)*
- *move_new (dest: smart_ptr<void>&; src: smart_ptr<void>)*
- *smart_ptr_clone (dest: smart_ptr<void>&; src: void?)*
- *smart_ptr_clone (dest: smart_ptr<void>&; src: smart_ptr<void>)*
- *smart_ptr_is_valid (dest: smart_ptr<void>) : bool*
- *smart_ptr_use_count (ptr: smart_ptr<void>) : uint*

add_ptr_ref

`_builtin::add_ptr_ref(src: auto(TT)?) : smart_ptr<TT>()`

Wraps a raw pointer `src` of type TT? into a `smart_ptr`<TT> by incrementing the reference count. Commonly used to bridge AST node fields (which are raw pointers like `Structure`?, `Enumeration`?) to API functions that expect `smart_ptr`<T>. The overload accepting `smart_ptr`<`auto`(TT)> adds an additional reference to an existing smart pointer, returning a new `smart_ptr`<TT> that shares ownership.

Arguments

- **src** : auto(TT)?

`_builtin::add_ptr_ref(src: smart_ptr<auto(TT)>) : smart_ptr<TT>()`

`_builtin::get_const_ptr(src: smart_ptr<auto(TT)>) : TT?()`

Extracts a constant raw pointer of type *TT?* from the given *smart_ptr<TT>*, without affecting reference counting.

Arguments

- **src** : smart_ptr<auto(TT)>

get_ptr

`_builtin::get_ptr(src: smart_ptr<auto(TT)> ==const) : TT?()`

Extracts a mutable raw pointer of type *TT?* from the given mutable *smart_ptr<TT>*, without affecting reference counting.

Arguments

- **src** : smart_ptr<auto(TT)>!

`_builtin::get_ptr(src: smart_ptr<auto(TT)> ==const) : TT?()`

---

move

`_builtin::move`(*dest: smart_ptr<void>&; src: void?*)

Moves a raw pointer *src* into the smart pointer *dest*, nullifying the previous contents of *dest* and transferring ownership of *src*.

Arguments

- **dest** : smart_ptr<void>& implicit
- **src** : void? implicit

`_builtin::move`(*dest: smart_ptr<void>&; src: smart_ptr<void>&*)

`_builtin::`**`move_new`**`(`*dest: smart_ptr<void>&; src: smart_ptr<void>*`)`

Moves a newly constructed smart pointer value *src* into *dest*, used to initialize a *smart_ptr* from a *new* expression.

Arguments

- **dest** : smart_ptr<void>& implicit
- **src** : smart_ptr<void> implicit

smart_ptr_clone

`_builtin::`**`smart_ptr_clone`**`(`*dest: smart_ptr<void>&; src: void?*`)`

Clones a raw pointer *src* into smart pointer *dest*, incrementing the internal reference count to share ownership.

Arguments

- **dest** : smart_ptr<void>& implicit
- **src** : void? implicit

`_builtin::`**`smart_ptr_clone`**`(`*dest: smart_ptr<void>&; src: smart_ptr<void>*`)`

`_builtin::`**`smart_ptr_is_valid(dest: smart_ptr<void>) : bool`**`()`

Checks whether the smart pointer *dest* holds a non-null reference to valid data, returning true if it does.

Arguments

- **dest** : smart_ptr<void> implicit

`_builtin::`**`smart_ptr_use_count(ptr: smart_ptr<void>) : uint`**`()`

Returns the current reference count of the object managed by *ptr*, indicating how many smart pointers share ownership.

Arguments

- **ptr** : smart_ptr<void> implicit

2.1.15 Macro infrastructure

- *is_compiling () : bool*
- *is_compiling_macros () : bool*
- *is_compiling_macros_in_module (name: string) : bool*
- *is_folding () : bool*
- *is_in_completion () : bool*
- *is_reporting_compilation_errors () : bool*

`_builtin::`**`is_compiling() : bool`**`()`

Returns *true* if the current context is in the process of being compiled, allowing compile-time logic to distinguish from runtime execution.

`_builtin::`**`is_compiling_macros() : bool`**`()`

Returns *true* if the current context is being compiled and the compiler is currently executing the macro pass.

`_builtin::`**`is_compiling_macros_in_module(name: string) : bool`**`()`

Returns *true* if the current context is being compiled during the macro pass and the compiler is processing the module specified by *name*.

Arguments

- **name** : string implicit

`_builtin::`**`is_folding() : bool`**`()`

Returns *true* if the compiler is currently performing its constant folding optimization pass.

`_builtin::`**`is_in_completion() : bool`**`()`

Returns *true* if the compiler is running in completion mode, generating lexical information for a text editor's code-completion system.

`_builtin::`**`is_reporting_compilation_errors() : bool`**`()`

Returns *true* if the context failed to compile and the inference pass is currently reporting compilation errors.

2.1.16 Profiler

- *collect_profile_info () : string*
- *dump_profile_info ()*
- *profile (count: int; category: string; block: block<():void>) : float*
- *reset_profiler ()*

`_builtin::`**`collect_profile_info() : string`**`()`

Collects profiling information gathered by the built-in line profiler and returns it as a formatted string containing execution counts and timing data.

`_builtin::`**`dump_profile_info`**`()`

Prints the execution counts and timing data for all lines collected by the built-in line profiler to the standard output.

`_builtin::`**`profile(count: int; category: string; block: block<():void>) : float`**`()`

Executes *block* a total of *count* times under the given *category* label, prints the timing, and returns the minimum elapsed time in seconds across all iterations.

Arguments

- **count** : int
- **category** : string implicit
- **block** : block<void> implicit

`_builtin::`**`reset_profiler`**`()`

Resets all counters and accumulated data in the built-in profiler to zero.

2.1.17 System infrastructure

- *aot_enabled () : bool*
- *breakpoint ()*
- *error (text: string)*
- *eval_main_loop (block: block<():void>)*
- *feint (text: string)*
- *get_das_root () : string*
- *is_in_aot () : bool*
- *is_intern_strings () : bool*
- *panic (text: string)*
- *print (text: string)*
- *sprint (value: any; flags: print_flags) : string*
- *sprint_json (value: any; humanReadable: bool) : string*
- *stackwalk (args: bool = true; vars: bool = true)*
- *terminate ()*
- *to_compiler_log (text: string)*
- *to_log (level: int; text: string)*

`_builtin::`**`aot_enabled() : bool`**`()`

Checks whether ahead-of-time (AOT) compilation is enabled for the current program and returns true if it is.

`_builtin::`**`breakpoint`**`()`

Triggers a debugger breakpoint by calling *os_debugbreakpoint*, which is a link-time dependency expected to be provided by the host application or debugger tool.

`_builtin::`**`error`**`(`*text: string*`)`

Outputs the string *text* to the context's error stream, similar to *print* but directed to the error output channel.

Arguments

- **text** : string implicit

`_builtin::`**`eval_main_loop`**`(`*block: block<():void>*`)`

Executes the application main loop by repeatedly invoking *block* until it returns *false*; on Emscripten targets, uses the platform-specific main loop mechanism instead.

Arguments

- **block** : block<void> implicit

`_builtin::`**`feint`**`(`*text: string*`)`

No-op replacement for *print*. Has the same signature and side-effect annotations as *print*, but intentionally does nothing. Use *feint* in tests where print-like behavior is needed to prevent the call from being optimized out, but no actual output is desired.

Arguments

- **text** : string implicit

`_builtin::get_das_root() : string()`

Returns the file-system path to the daslang root directory, where *daslib* and other standard libraries are located.

`_builtin::is_in_aot() : bool()`

Returns *true* if the compiler is currently generating ahead-of-time (AOT) compiled code.

`_builtin::is_intern_strings() : bool()`

Returns *true* if string interning is enabled in the current context, meaning identical strings share the same memory.

`_builtin::panic(`*text: string*`)`

will cause panic. The program will be terminated if there is no recover. Panic is not an error handling mechanism and can not be used as such. It is indeed panic, fatal error. It is not supposed that program can completely correctly recover from panic, recover construction is provided so program can try to correctly shut-down or report fatal error. If there is no recover within the script, it will be called in calling eval (in C++ callee code).

Arguments

- **text** : string implicit

`_builtin::print(`*text: string*`)`

Outputs *text* to the current context's log, typically printing to standard output.

Arguments

- **text** : string implicit

`_builtin::sprint(value: any; flags: print_flags) : string()`

Converts *value* to its string representation using the specified *flags* to control formatting, and returns the result as a string.

Arguments

- **value** : any
- **flags** : *print_flags*

`_builtin::sprint_json(value: any; humanReadable: bool) : string()`

Serializes *value* directly to a JSON string, bypassing intermediate representation for speed; set *humanReadable* to true for indented output.

Arguments

- **value** : any
- **humanReadable** : bool

`_builtin::stackwalk(`*args: bool = true; vars: bool = true*`)`

Prints the current call stack to the log; set *args* to include function arguments and *vars* to include local variable values in the output.

Arguments

- **args** : bool
- **vars** : bool

`_builtin::`**`terminate`**`()`

Immediately terminates execution of the current daslang context.

`_builtin::`**`to_compiler_log`**`(`*text: string*`)`

Outputs *text* to the compiler's log stream, typically used from within macro code during compilation.

Arguments

- **text** : string implicit

`_builtin::`**`to_log`**`(`*level: int; text: string*`)`

Outputs *text* to the logging infrastructure at the specified *level* (e.g. LOG_INFO, LOG_ERROR), rather than to standard output.

Arguments

- **level** : int
- **text** : string implicit

2.1.18 Memory manipulation

- *hash (value: int8) : uint64*
- *hash (data: any) : uint64*
- *hash (data: string) : uint64*
- *hash (value: int) : uint64*
- *hash (value: int16) : uint64*
- *hash (value: uint8) : uint64*
- *hash (value: uint16) : uint64*
- *hash (value: double) : uint64*
- *hash (value: int64) : uint64*
- *hash (value: uint) : uint64*
- *hash (value: uint64) : uint64*
- *hash (value: void?) : uint64*
- *hash (value: float) : uint64*
- *hash (value: das_string) : uint64*
- *intptr (p: smart_ptr<auto>) : uint64*
- *intptr (p: void?) : uint64*
- *lock_data (a: array<auto(TT)> ==const|array<auto(TT)> const# ==const; blk: block<(p:TT const?#;s:int):auto>) : auto*
- *lock_data (var a: array<auto(TT)> ==const|array<auto(TT)># ==const; blk: block<(var p:TT?#;s:int):auto>) : auto*
- *map_to_array (data: void?; len: int; blk: block<(var arg:array<auto(TT)>#):auto>) : auto*
- *map_to_ro_array (data: void?; len: int; blk: block<(arg:array<auto(TT)>#):auto>) : auto*

- *memcmp (left: void?; right: void?; size: int) : int*
- *memcpy (left: void?; right: void?; size: int)*
- *set_variant_index (variant: variant<>; index: int)*
- *variant_index (arg0: variant<>) : int*

hash

`_builtin::hash(value: int8) : uint64()`

Computes a 64-bit FNV-1a hash of the given *int8* value and returns it as *uint64*.

Arguments

- **value** : int8

`_builtin::hash(data: any) : uint64()`

`_builtin::hash(data: string) : uint64()`

`_builtin::hash(value: int) : uint64()`

`_builtin::hash(value: int16) : uint64()`

`_builtin::hash(value: uint8) : uint64()`

`_builtin::hash(value: uint16) : uint64()`

`_builtin::hash(value: double) : uint64()`

`_builtin::hash(value: int64) : uint64()`

`_builtin::hash(value: uint) : uint64()`

`_builtin::hash(value: uint64) : uint64()`

`_builtin::hash(value: void?) : uint64()`

`_builtin::hash(value: float) : uint64()`

`_builtin::hash(value: das_string) : uint64()`

---

intptr

`_builtin::intptr(p: smart_ptr<auto>) : uint64()`

Converts a *smart_ptr p* to its *uint64* integer representation, useful for pointer arithmetic or serialization.

Arguments

- **p** : smart_ptr<auto>

`_builtin::intptr(p: void?) : uint64()`

---

lock_data

`_builtin::lock_data(a: array<auto(TT)> ==const|array<auto(TT)> const# ==const; blk: block<(p:TT const?#`

Locks a constant array and invokes *blk* with a read-only pointer *p* to the array's contiguous data and its size *s*, allowing direct memory-level read access.

Arguments

- **a** : option<array<auto(TT)>!| array<auto(TT)>#!>
- **blk** : block<(p:TT?#;s:int):auto>

`_builtin::lock_data(a: array<auto(TT)> ==const|array<auto(TT)># ==const; blk: block<(var p:TT?#;s:int):`

`_builtin::map_to_array(data: void?; len: int; blk: block<(var arg:array<auto(TT)>#):auto>) : auto()`

Warning: This is unsafe operation.

Constructs a temporary mutable array of type *TT* over raw memory at *data* with *len* elements, and passes it to *blk* without copying the underlying data.

Arguments

- **data** : void?
- **len** : int
- **blk** : block<(arg:array<auto(TT)>#):auto>

`_builtin::map_to_ro_array(data: void?; len: int; blk: block<(arg:array<auto(TT)>#):auto>) : auto()`

Warning: This is unsafe operation.

Constructs a temporary read-only array of type *TT* over raw memory at *data* with *len* elements, and passes it to *blk* without copying the underlying data.

Arguments

- **data** : void?
- **len** : int
- **blk** : block<(arg:array<auto(TT)>#):auto>

`_builtin::memcmp(left: void?; right: void?; size: int) : int()`

Warning: This is unsafe operation.

Compares *size* bytes of memory at *left* and *right*, returning -1 if *left* is less, 1 if *left* is greater, or 0 if both regions are identical.

Arguments

- **left** : void? implicit

- **right** : void? implicit
- **size** : int

`_builtin::`**`memcpy`**(*left: void?; right: void?; size: int*)

Warning: This is unsafe operation.

Copies *size* bytes of memory from the address pointed to by *right* into the address pointed to by *left*.

Arguments

- **left** : void? implicit
- **right** : void? implicit
- **size** : int

`_builtin::`**`set_variant_index`**(*variant: variant<>; index: int*)

Warning: This is unsafe operation.

Overwrites the internal type discriminator of *variant* to *index*, changing which alternative the variant is considered to hold.

Arguments

- **variant** : variant<> implicit
- **index** : int

`_builtin::`**`variant_index(arg0: variant<>) : int`**`()`

Returns the zero-based index indicating which alternative the variant currently holds.

Arguments

- **arg0** : variant<> implicit

2.1.19 Binary serializer

- *binary_load (var obj: auto; data: array<uint8>) : auto*
- *binary_save (obj: auto; subexpr: block<(data:array<uint8>#):void>) : auto*

`_builtin::`**`binary_load(obj: auto; data: array<uint8>) : auto`**`()`

Deserializes *obj* from the binary representation stored in *data* (an array of uint8 bytes). Obsolete — use *daslib/archive* instead.

Arguments

- **obj** : auto
- **data** : array<uint8> implicit

`_builtin::`**`binary_save(obj: auto; subexpr: block<(data:array<uint8>#):void>) : auto`**`()`

Serializes *obj* into a binary representation and passes the resulting uint8 byte array to the block *subexpr*. Obsolete — use *daslib/archive* instead.

Arguments

- **obj** : auto
- **subexpr** : block<(data:array<uint8>#):void>

2.1.20 Path and command line

- *get_command_line_arguments () : array<string>*

`_builtin::`**`get_command_line_arguments() : array<string>`**`()`

Returns an array of strings containing the command-line arguments passed to the program.

2.1.21 Time and date

- *get_clock () : clock*
- *get_time_nsec (ref: int64) : int64*
- *get_time_usec (ref: int64) : int*
- *mktime (year: int; month: int; mday: int; hour: int; min: int; sec: int) : clock*
- *ref_time_ticks () : int64*

`_builtin::`**`get_clock() : clock`**`()`

Returns the current calendar time as a *clock* value representing the number of seconds since 00:00 UTC, January 1, 1970 (the Unix epoch).

`_builtin::`**`get_time_nsec(ref: int64) : int64`**`()`

Computes the elapsed time in nanoseconds since the reference point *ref*, which is typically obtained from *ref_time_ticks*.

Arguments

- **ref** : int64

`_builtin::`**`get_time_usec(ref: int64) : int`**`()`

Computes the elapsed time in microseconds since the reference point *ref*, which is typically obtained from *ref_time_ticks*.

Arguments

- **ref** : int64

`_builtin::`**`mktime(year: int; month: int; mday: int; hour: int; min: int; sec: int) : clock`**`()`

Converts the calendar date and time specified by *year*, *month*, *mday*, *hour*, *min*, and *sec* into a *clock* value representing time since epoch.

Arguments

- **year** : int
- **month** : int

- **mday** : int
- **hour** : int
- **min** : int
- **sec** : int

`_builtin::`**`ref_time_ticks() : int64`**`()`

Captures the current high-resolution time in ticks, suitable for measuring elapsed intervals with *get_time_usec*.

2.1.22 Platform queries

- *das_is_dll_build () : bool*
- *get_architecture_name () : string*
- *get_context_share_counter () : uint64*
- *get_platform_name () : string*

`_builtin::`**`das_is_dll_build() : bool`**`()`

Checks whether the current build is configured as a DLL (dynamic library) build, which determines if daslib symbols are available for the JIT compiler.

`_builtin::`**`get_architecture_name() : string`**`()`

Returns the name of the CPU architecture the program is running on, such as *"x86_64"*, *"x86"*, *"arm64"*, *"arm"*, *"wasm32"*, or *"unknown"*.

`_builtin::`**`get_context_share_counter() : uint64`**`()`

Returns the use-count of the shared context, which is incremented each time a thread accesses it; useful for tracking concurrent context usage.

`_builtin::`**`get_platform_name() : string`**`()`

Returns the name of the operating system the program is running on, such as *"windows"*, *"linux"*, *"darwin"*, *"emscripten"*, or *"unknown"*.

2.1.23 String formatting

- *fmt (format: string; value: uint8) : string*
- *fmt (format: string; value: int8) : string*
- *fmt (format: string; value: uint16) : string*
- *fmt (format: string; value: int16) : string*
- *fmt (format: string; value: int64) : string*
- *fmt (format: string; value: uint) : string*
- *fmt (format: string; value: uint64) : string*
- *fmt (format: string; value: int) : string*
- *fmt (format: string; value: double) : string*
- *fmt (format: string; value: float) : string*

fmt

`_builtin::fmt(format: string; value: uint8) : string()`

Formats a *uint8* value as a string using the given *format* specifier (following libfmt / C++20 *std::format* syntax).

Arguments

- **format** : string implicit
- **value** : uint8

`_builtin::fmt(format: string; value: int8) : string()`

`_builtin::fmt(format: string; value: uint16) : string()`

`_builtin::fmt(format: string; value: int16) : string()`

`_builtin::fmt(format: string; value: int64) : string()`

`_builtin::fmt(format: string; value: uint) : string()`

`_builtin::fmt(format: string; value: uint64) : string()`

`_builtin::fmt(format: string; value: int) : string()`

`_builtin::fmt(format: string; value: double) : string()`

`_builtin::fmt(format: string; value: float) : string()`

2.1.24 Argument consumption

- *consume_argument (var a: auto(TT)&) : TT&*

`_builtin::consume_argument(a: auto(TT)&) : TT&()`

Marks argument *a* as consumed, signaling to the compiler that it will not be used after this call, which enables move optimizations and avoids unnecessary clones. Equivalent to the *<-arg* syntax.

Arguments

- **a** : auto(TT)&

2.1.25 Lock checking

- *lock_count (array: array<anything>) : int*
- *set_verify_array_locks (array: array<anything>; check: bool) : bool*
- *set_verify_table_locks (table: table<anything, anything>; check: bool) : bool*

`_builtin::lock_count(array: array<anything>) : int()`

Returns the current internal lock count for the given *array*, indicating how many active locks prevent it from being resized.

Arguments

- **array** : array implicit

`_builtin::set_verify_array_locks(array: array<anything>; check: bool) : bool()`

Warning: This is unsafe operation.

Enables or disables runtime lock verification for the given *array*; when *check* is false, lock safety checks are skipped as a performance optimization.

Arguments

- **array** : array implicit
- **check** : bool

`_builtin::set_verify_table_locks(table: table<anything, anything>; check: bool) : bool()`

Warning: This is unsafe operation.

Enables or disables runtime lock verification for the given *table*; when *check* is false, lock safety checks are skipped as a performance optimization.

Arguments

- **table** : table implicit
- **check** : bool

2.1.26 Lock checking internals

- *_at_with_lockcheck (var Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT&*
- *_move_with_lockcheck (var a: auto(valA)&; var b: auto(valB)&) : auto*
- *_return_with_lockcheck (var a: auto(valT)& ==const) : auto&*
- *_return_with_lockcheck (a: auto(valT) const& ==const) : auto&*

`_builtin::_at_with_lockcheck(Tab: table<auto(keyT), auto(valT)>; at: keyT|keyT#) : valT&()`

Looks up and returns a reference to the element at key *at* in the table *Tab*, while verifying that *Tab* and its lockable sub-elements are not locked.

Arguments

- **Tab** : table<auto(keyT);auto(valT)>
- **at** : option<keyT| keyT#>

`_builtin::_move_with_lockcheck(a: auto(valA)&; b: auto(valB)&) : auto()`

Moves the contents of *b* into *a*, verifying that neither *a* nor *b* (nor any of their lockable sub-elements) is currently locked.

Arguments

- **a** : auto(valA)&
- **b** : auto(valB)&

_return_with_lockcheck

`_builtin::_return_with_lockcheck(a: auto(valT)& ==const) : auto&()`

Passes through and returns a mutable reference to *a*, verifying that *a* and all of its lockable sub-elements are not currently locked.

Arguments

- **a** : auto(valT)&!

`_builtin::_return_with_lockcheck(a: auto(valT) const& ==const) : auto&()`

2.1.27 Bit operations

- *__bit_set (value: bitfield&; mask: bitfield; on: bool)*
- *__bit_set (value: bitfield8:uint8<>&; mask: bitfield8:uint8<>; on: bool)*
- *__bit_set (value: bitfield64:uint64<>&; mask: bitfield64:uint64<>; on: bool)*
- *__bit_set (value: bitfield16:uint16<>&; mask: bitfield16:uint16<>; on: bool)*
- *clz (bits: uint64) : uint64*
- *clz (bits: uint) : uint*
- *ctz (bits: uint64) : uint64*
- *ctz (bits: uint) : uint*
- *mul128 (a: uint64; b: uint64) : urange64*
- *popcnt (bits: uint64) : uint64*
- *popcnt (bits: uint) : uint*

__bit_set

`_builtin::__bit_set(value: bitfield&; mask: bitfield; on: bool)`

Sets or clears the bits specified by *mask* in the 16-bit bitfield *value*, turning them on if *on* is true or off if *on* is false.

Arguments

- **value** : bitfield<>& implicit
- **mask** : bitfield<>
- **on** : bool

`_builtin::__bit_set(value: bitfield8:uint8<>&; mask: bitfield8:uint8<>; on: bool)`

`_builtin::__bit_set(value: bitfield64:uint64<>&; mask: bitfield64:uint64<>; on: bool)`

`_builtin::__bit_set(value: bitfield16:uint16<>&; mask: bitfield16:uint16<>; on: bool)`

clz

`_builtin::clz(bits: uint64) : uint64()`

Counts the number of leading zero bits in the 64-bit unsigned integer *bits*, returning 64 if the value is zero.

Arguments

- **bits** : uint64

`_builtin::clz(bits: uint) : uint()`

---

ctz

`_builtin::ctz(bits: uint64) : uint64()`

Counts the number of trailing zero bits in the 64-bit unsigned integer *bits*, returning 64 if the value is zero.

Arguments

- **bits** : uint64

`_builtin::ctz(bits: uint) : uint()`

`_builtin::mul128(a: uint64; b: uint64) : urange64()`

Multiplies two 64-bit unsigned integers *a* and *b*, returning the full 128-bit result as a *urange64* containing the low and high 64-bit halves.

Arguments

- **a** : uint64
- **b** : uint64

popcnt

`_builtin::popcnt(bits: uint64) : uint64()`

Counts and returns the number of set (1) bits in the 64-bit unsigned integer *bits*.

Arguments

- **bits** : uint64

`_builtin::popcnt(bits: uint) : uint()`

2.1.28 Intervals

- *interval (arg0: uint; arg1: uint) : urange*
- *interval (arg0: int; arg1: int) : range*
- *interval (arg0: uint64; arg1: uint64) : urange64*
- *interval (arg0: int64; arg1: int64) : range64*

interval

`_builtin::`**`interval(arg0: uint; arg1: uint) : urange`**`()`

Constructs a *urange* value from the two *uint* endpoints *arg0* (inclusive) and *arg1* (exclusive).

Arguments

- **arg0** : uint
- **arg1** : uint

`_builtin::`**`interval(arg0: int; arg1: int) : range`**`()`

`_builtin::`**`interval(arg0: uint64; arg1: uint64) : urange64`**`()`

`_builtin::`**`interval(arg0: int64; arg1: int64) : range64`**`()`

2.1.29 RTTI

- *class_rtti_size (ptr: void?) : int*

`_builtin::`**`class_rtti_size(ptr: void?) : int`**`()`

Examines the RTTI (runtime type information) associated with the class at *ptr* and returns the size in bytes of its TypeInfo structure.

Arguments

- **ptr** : void? implicit

2.1.30 Lock verification

- *set_verify_context_locks (check: bool) : bool*

`_builtin::`**`set_verify_context_locks(check: bool) : bool`**`()`

Warning: This is unsafe operation.

Enables or disables runtime lock verification for all arrays and tables in the current context; returns the previous verification state.

Arguments

- **check** : bool

2.1.31 Initialization and finalization

- *using (arg0: block<(das_string):void>)*

`_builtin::`**`using`**(*arg0: block<(das_string):void>*)

Creates a temporary *das_string* and passes it to the block, automatically managing its lifetime for the duration of the call.

Arguments

- **arg0** : block<(*das_string*):void> implicit

2.1.32 Algorithms

- *count (start: int = 0; step: int = 1) : iterator<int>*
- *iter_range (foo: auto) : auto*
- *swap (var a: auto(TT)&; var b: auto(TT)&) : auto*
- *ucount (start: uint = 0x0; step: uint = 0x1) : iterator<uint>*

`_builtin::`**`count(start: int = 0; step: int = 1) : iterator<int>`**`()`

Creates an infinite iterator that yields integer values starting from *start* and incrementing by *step* on each iteration, intended for use as a counter alongside other sequences in a *for* loop.

Arguments

- **start** : int
- **step** : int

`_builtin::`**`iter_range(foo: auto) : auto`**`()`

Creates a *range* from *0* to the length of the given iterable *foo*, useful for index-based iteration over containers.

Arguments

- **foo** : auto

`_builtin::`**`swap(a: auto(TT)&; b: auto(TT)&) : auto`**`()`

Exchanges the values of *a* and *b* in place, leaving each variable holding the other's former value.

Arguments

- **a** : auto(TT)&
- **b** : auto(TT)&

`_builtin::`**`ucount(start: uint = 0x0; step: uint = 0x1) : iterator<uint>`**`()`

Creates an infinite iterator over unsigned integers beginning at *start* and incrementing by *step* on each iteration.

Arguments

- **start** : uint
- **step** : uint

2.1.33 Memset

- *memset128 (left: void?; value: uint4; count: int)*
- *memset16 (left: void?; value: uint16; count: int)*
- *memset32 (left: void?; value: uint; count: int)*
- *memset64 (left: void?; value: uint64; count: int)*
- *memset8 (left: void?; value: uint8; count: int)*

`_builtin::`**`memset128`**`(`*left: void?; value: uint4; count: int*`)`

> **Warning:** This is unsafe operation.

Fills memory at *left* with *count* copies of the 128-bit *uint4* vector *value*.

Arguments

- **left** : void? implicit
- **value** : uint4
- **count** : int

`_builtin::`**`memset16`**`(`*left: void?; value: uint16; count: int*`)`

> **Warning:** This is unsafe operation.

Fills memory at *left* with *count* copies of the 16-bit *value*.

Arguments

- **left** : void? implicit
- **value** : uint16
- **count** : int

`_builtin::`**`memset32`**`(`*left: void?; value: uint; count: int*`)`

> **Warning:** This is unsafe operation.

Fills memory at *left* with *count* copies of the 32-bit *value*.

Arguments

- **left** : void? implicit
- **value** : uint
- **count** : int

`_builtin::`**`memset64`**`(`*left: void?; value: uint64; count: int*`)`

> **Warning:** This is unsafe operation.

Fills memory at *left* with *count* copies of the 64-bit *value*.

Arguments

- **left** : void? implicit
- **value** : uint64
- **count** : int

`_builtin::`**`memset8`**`(`*left: void?; value: uint8; count: int*`)`

Warning: This is unsafe operation.

Fills memory at *left* with *count* copies of the 8-bit *value*, equivalent to the C *memset* function.

Arguments

- **left** : void? implicit
- **value** : uint8
- **count** : int

2.1.34 Malloc

- *free (ptr: void?)*
- *malloc (size: uint64) : void?*
- *malloc_usable_size (ptr: void?) : uint64*

`_builtin::`**`free`**`(`*ptr: void?*`)`

Warning: This is unsafe operation.

Frees memory previously allocated with *malloc*, following C-style manual memory management semantics.

Arguments

- **ptr** : void? implicit

`_builtin::`**`malloc(size: uint64) : void?`**`()`

Warning: This is unsafe operation.

Allocates a block of uninitialized memory of the specified *size* in bytes, C-style, and returns a raw pointer to it; must be freed with *free*.

Arguments

- **size** : uint64

`_builtin::`**`malloc_usable_size(ptr: void?) : uint64`**`()`

Warning: This is unsafe operation.

Returns the usable size in bytes of the memory block pointed to by *ptr*, as reported by the underlying allocator.

Arguments

- **ptr** : void? implicit

2.1.35 Compilation and AOT

- *compiling_file_name () : string*
- *compiling_module_name () : string*
- *reset_aot ()*
- *set_aot ()*

`_builtin::`**`compiling_file_name() : string`**`()`

Returns the file name of the source file currently being compiled, useful for compile-time metaprogramming and diagnostics.

`_builtin::`**`compiling_module_name() : string`**`()`

Returns the name of the module currently being compiled, useful for compile-time metaprogramming and diagnostics.

`_builtin::`**`reset_aot`**`()`

Notifies the compiler that ahead-of-time code generation has finished, restoring normal compilation mode.

`_builtin::`**`set_aot`**`()`

Notifies the compiler that ahead-of-time code generation is now in progress.

2.2 Math library

The MATH module contains floating point math functions and constants (trigonometry, exponentials, clamping, interpolation, noise, and vector/matrix operations). Floating point math in general is not bit-precise: the compiler may optimize permutations, replace divisions with multiplications, and some functions are not bit-exact. Use `double` precision types when exact results are required.

All functions and symbols are in “math” module, use require to get access to it.

```
require math
```

Example:

```
require math

    [export]
    def main() {
        print("sin(PI/2) = {sin(PI / 2.0)}\n")
        print("cos(0)    = {cos(0.0)}\n")
        print("sqrt(16)  = {sqrt(16.0)}\n")
        print("abs(-5)   = {abs(-5)}\n")
```

(continues on next page)

(continued from previous page)

```
        print("clamp(15, 0, 10) = {clamp(15, 0, 10)}\n")
        print("min(3, 7) = {min(3, 7)}\n")
        print("max(3, 7) = {max(3, 7)}\n")
        let v = float3(1, 0, 0)
        print("length = {length(v)}\n")
    }
    // output:
    // sin(PI/2) = 1
    // cos(0)    = 1
    // sqrt(16)  = 4
    // abs(-5)   = 5
    // clamp(15, 0, 10) = 10
    // min(3, 7) = 3
    // max(3, 7) = 7
    // length = 1
```

2.2.1 Constants

`PI = 3.1415927f`

The single-precision float constant pi (3.14159265…), representing the ratio of a circle's circumference to its diameter.

`DBL_PI = 3.141592653589793lf`

The double-precision constant pi (3.141592653589793…), representing the ratio of a circle's circumference to its diameter.

`FLT_EPSILON = 1.1920929e-07f`

The smallest single-precision float value epsilon such that 1.0f + epsilon != 1.0f, approximately 1.1920929e-7.

`DBL_EPSILON = 2.220446049250313e-16lf`

The smallest double-precision value epsilon such that 1.0 + epsilon != 1.0, approximately 2.2204460492503131e-16.

2.2.2 Handled structures

`math::float4x4`

floating point matrix with 4 rows and 4 columns

Fields

- **x** : float4 - 0th row
- **y** : float4 - 1st row
- **z** : float4 - 2nd row
- **w** : float4 - 3rd row

`math::float3x4`

floating point matrix with 4 rows and 3 columns

Fields

- **x** : float3 - 0th row
- **y** : float3 - 1st row
- **z** : float3 - 2nd row
- **w** : float3 - 3rd row

`math::float3x3`

floating point matrix with 3 rows and 3 columns

Fields

- **x** : float3 - 0th row
- **y** : float3 - 1st row
- **z** : float3 - 2nd row

2.2.3 all numerics (uint*, int*, float*, double)

- *max (x: int; y: int) : int*
- *max (x: uint64; y: uint64) : uint64*
- *max (x: int2; y: int2) : int2*
- *max (x: double; y: double) : double*
- *max (x: int4; y: int4) : int4*
- *max (x: int3; y: int3) : int3*
- *max (x: float3; y: float3) : float3*
- *max (x: int64; y: int64) : int64*
- *max (x: uint2; y: uint2) : uint2*
- *max (x: uint; y: uint) : uint*
- *max (x: uint3; y: uint3) : uint3*
- *max (x: float4; y: float4) : float4*
- *max (x: uint4; y: uint4) : uint4*
- *max (x: float2; y: float2) : float2*
- *max (x: float; y: float) : float*
- *min (x: uint3; y: uint3) : uint3*
- *min (x: uint2; y: uint2) : uint2*
- *min (x: int4; y: int4) : int4*
- *min (x: int; y: int) : int*
- *min (x: float3; y: float3) : float3*
- *min (x: float4; y: float4) : float4*
- *min (x: float2; y: float2) : float2*
- *min (x: float; y: float) : float*
- *min (x: int2; y: int2) : int2*

- *min (x: int3; y: int3) : int3*
- *min (x: uint; y: uint) : uint*
- *min (x: uint64; y: uint64) : uint64*
- *min (x: uint4; y: uint4) : uint4*
- *min (x: double; y: double) : double*
- *min (x: int64; y: int64) : int64*

max

`math::max(x: int; y: int) : int()`

Returns the component-wise maximum of two values, supporting scalar double, float, int, int64, uint, uint64 and vector float2, float3, float4 types.

Arguments

- **x** : int
- **y** : int

`math::max(x: uint64; y: uint64) : uint64()`

`math::max(x: int2; y: int2) : int2()`

`math::max(x: double; y: double) : double()`

`math::max(x: int4; y: int4) : int4()`

`math::max(x: int3; y: int3) : int3()`

`math::max(x: float3; y: float3) : float3()`

`math::max(x: int64; y: int64) : int64()`

`math::max(x: uint2; y: uint2) : uint2()`

`math::max(x: uint; y: uint) : uint()`

`math::max(x: uint3; y: uint3) : uint3()`

`math::max(x: float4; y: float4) : float4()`

`math::max(x: uint4; y: uint4) : uint4()`

`math::max(x: float2; y: float2) : float2()`

`math::max(x: float; y: float) : float()`

min

`math::min(x: uint3; y: uint3) : uint3()`

Returns the component-wise minimum of two values, supporting scalar double, float, int, int64, uint, uint64 and vector float2, float3, float4 types.

Arguments

- **x** : uint3
- **y** : uint3

`math::min(x: uint2; y: uint2) : uint2()`

`math::min(x: int4; y: int4) : int4()`

`math::min(x: int; y: int) : int()`

`math::min(x: float3; y: float3) : float3()`

`math::min(x: float4; y: float4) : float4()`

`math::min(x: float2; y: float2) : float2()`

`math::min(x: float; y: float) : float()`

`math::min(x: int2; y: int2) : int2()`

`math::min(x: int3; y: int3) : int3()`

`math::min(x: uint; y: uint) : uint()`

`math::min(x: uint64; y: uint64) : uint64()`

`math::min(x: uint4; y: uint4) : uint4()`

`math::min(x: double; y: double) : double()`

`math::min(x: int64; y: int64) : int64()`

2.2.4 float* and double

- *abs (x: float3) : float3*
- *abs (x: uint) : uint*
- *abs (x: uint64) : uint64*
- *abs (x: int2) : int2*
- *abs (x: int4) : int4*
- *abs (x: int3) : int3*
- *abs (x: int64) : int64*
- *abs (x: uint2) : uint2*
- *abs (x: int) : int*
- *abs (x: uint4) : uint4*
- *abs (x: uint3) : uint3*

- *abs (x: double) : double*
- *abs (x: float) : float*
- *abs (x: float4) : float4*
- *abs (x: float2) : float2*
- *acos (x: float4) : float4*
- *acos (x: float2) : float2*
- *acos (x: float) : float*
- *acos (x: float3) : float3*
- *acos (x: double) : double*
- *asin (x: float) : float*
- *asin (x: double) : double*
- *asin (x: float3) : float3*
- *asin (x: float4) : float4*
- *asin (x: float2) : float2*
- *atan (x: float2) : float2*
- *atan (x: float4) : float4*
- *atan (x: double) : double*
- *atan (x: float3) : float3*
- *atan (x: float) : float*
- *atan2 (y: float3; x: float3) : float3*
- *atan2 (y: float2; x: float2) : float2*
- *atan2 (y: float; x: float) : float*
- *atan2 (y: float4; x: float4) : float4*
- *atan2 (y: double; x: double) : double*
- *ceil (x: float4) : float4*
- *ceil (x: float3) : float3*
- *ceil (x: float2) : float2*
- *ceil (x: float) : float*
- *cos (x: double) : double*
- *cos (x: float4) : float4*
- *cos (x: float3) : float3*
- *cos (x: float) : float*
- *cos (x: float2) : float2*
- *exp (x: float4) : float4*
- *exp (x: double) : double*
- *exp (x: float2) : float2*

- *exp (x: float) : float*
- *exp (x: float3) : float3*
- *exp2 (x: float4) : float4*
- *exp2 (x: float3) : float3*
- *exp2 (x: float2) : float2*
- *exp2 (x: double) : double*
- *exp2 (x: float) : float*
- *floor (x: float) : float*
- *floor (x: float2) : float2*
- *floor (x: float4) : float4*
- *floor (x: float3) : float3*
- *is_finite (x: float) : bool*
- *is_finite (x: double) : bool*
- *is_nan (x: float) : bool*
- *is_nan (x: double) : bool*
- *log (x: float4) : float4*
- *log (x: float3) : float3*
- *log (x: float2) : float2*
- *log (x: double) : double*
- *log (x: float) : float*
- *log2 (x: double) : double*
- *log2 (x: float4) : float4*
- *log2 (x: float3) : float3*
- *log2 (x: float2) : float2*
- *log2 (x: float) : float*
- *pow (x: double; y: double) : double*
- *pow (x: float4; y: float4) : float4*
- *pow (x: float3; y: float3) : float3*
- *pow (x: float; y: float) : float*
- *pow (x: float2; y: float2) : float2*
- *rcp (x: double) : double*
- *rcp (x: float4) : float4*
- *rcp (x: float3) : float3*
- *rcp (x: float2) : float2*
- *rcp (x: float) : float*
- *safe_acos (x: double) : double*

- *safe_acos (x: float2) : float2*
- *safe_acos (x: float3) : float3*
- *safe_acos (x: float) : float*
- *safe_acos (x: float4) : float4*
- *safe_asin (x: float3) : float3*
- *safe_asin (x: float4) : float4*
- *safe_asin (x: double) : double*
- *safe_asin (x: float2) : float2*
- *safe_asin (x: float) : float*
- *saturate (x: float4) : float4*
- *saturate (x: float3) : float3*
- *saturate (x: float2) : float2*
- *saturate (x: float) : float*
- *sign (x: uint) : uint*
- *sign (x: uint2) : uint2*
- *sign (x: int4) : int4*
- *sign (x: float4) : float4*
- *sign (x: int64) : int64*
- *sign (x: double) : double*
- *sign (x: float3) : float3*
- *sign (x: uint4) : uint4*
- *sign (x: float) : float*
- *sign (x: float2) : float2*
- *sign (x: uint64) : uint64*
- *sign (x: int3) : int3*
- *sign (x: int) : int*
- *sign (x: int2) : int2*
- *sign (x: uint3) : uint3*
- *sin (x: double) : double*
- *sin (x: float4) : float4*
- *sin (x: float3) : float3*
- *sin (x: float2) : float2*
- *sin (x: float) : float*
- *sincos (x: float; s: float&; c: float&)*
- *sincos (x: double; s: double&; c: double&)*
- *sqrt (x: double) : double*

- *sqrt (x: float4) : float4*
- *sqrt (x: float3) : float3*
- *sqrt (x: float2) : float2*
- *sqrt (x: float) : float*
- *tan (x: double) : double*
- *tan (x: float4) : float4*
- *tan (x: float2) : float2*
- *tan (x: float) : float*
- *tan (x: float3) : float3*

abs

`math::abs(x: float3) : float3()`

Returns the absolute value of the argument, computed component-wise for float2, float3, float4, int2, int3, and int4 vector types, and per-element for scalar float, double, int, and int64 types.

Arguments

- **x** : float3

`math::abs(x: uint) : uint()`

`math::abs(x: uint64) : uint64()`

`math::abs(x: int2) : int2()`

`math::abs(x: int4) : int4()`

`math::abs(x: int3) : int3()`

`math::abs(x: int64) : int64()`

`math::abs(x: uint2) : uint2()`

`math::abs(x: int) : int()`

`math::abs(x: uint4) : uint4()`

`math::abs(x: uint3) : uint3()`

`math::abs(x: double) : double()`

`math::abs(x: float) : float()`

`math::abs(x: float4) : float4()`

`math::abs(x: float2) : float2()`

acos

`math::acos(x: float4) : float4()`

Returns the arccosine of x in radians; the input must be in the range [-1, 1] and the result is in the range [0, pi]; works with float and double.

Arguments

- **x** : float4

`math::acos(x: float2) : float2()`

`math::acos(x: float) : float()`

`math::acos(x: float3) : float3()`

`math::acos(x: double) : double()`

---

asin

`math::asin(x: float) : float()`

Returns the arcsine of x in radians; the input must be in the range [-1, 1] and the result is in the range [-pi/2, pi/2]; works with float and double.

Arguments

- **x** : float

`math::asin(x: double) : double()`

`math::asin(x: float3) : float3()`

`math::asin(x: float4) : float4()`

`math::asin(x: float2) : float2()`

---

atan

`math::atan(x: float2) : float2()`

Returns the arctangent of x in radians, with the result in the range [-pi/2, pi/2]; works with float and double.

Arguments

- **x** : float2

`math::atan(x: float4) : float4()`

`math::atan(x: double) : double()`

`math::atan(x: float3) : float3()`

`math::atan(x: float) : float()`

---

atan2

`math::atan2(y: float3; x: float3) : float3()`

Returns the arctangent of y/x in radians, using the signs of both arguments to determine the correct quadrant; the result is in the range [-pi, pi]; works with float and double.

Arguments

- **y** : float3
- **x** : float3

`math::atan2(y: float2; x: float2) : float2()`

`math::atan2(y: float; x: float) : float()`

`math::atan2(y: float4; x: float4) : float4()`

`math::atan2(y: double; x: double) : double()`

---

ceil

`math::ceil(x: float4) : float4()`

Returns the smallest integral value not less than x (rounds toward positive infinity), computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float4

`math::ceil(x: float3) : float3()`

`math::ceil(x: float2) : float2()`

`math::ceil(x: float) : float()`

---

cos

`math::cos(x: double) : double()`

Returns the cosine of x, where x is specified in radians; works with float and double.

Arguments

- **x** : double

`math::cos(x: float4) : float4()`

`math::cos(x: float3) : float3()`

`math::cos(x: float) : float()`

`math::cos(x: float2) : float2()`

---

exp

`math::exp(x: float4) : float4()`

Returns e raised to the power of x (the base-e exponential), computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float4

`math::exp(x: double) : double()`

`math::exp(x: float2) : float2()`

`math::exp(x: float) : float()`

`math::exp(x: float3) : float3()`

---

exp2

`math::exp2(x: float4) : float4()`

Returns 2 raised to the power of x, computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float4

`math::exp2(x: float3) : float3()`

`math::exp2(x: float2) : float2()`

`math::exp2(x: double) : double()`

`math::exp2(x: float) : float()`

---

floor

`math::floor(x: float) : float()`

Returns the largest integral value not greater than x (rounds toward negative infinity), computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float

`math::floor(x: float2) : float2()`

`math::floor(x: float4) : float4()`

`math::floor(x: float3) : float3()`

---

is_finite

math::**is_finite(x: float) : bool**()

Returns true if x is a finite value (not infinity and not NaN), checked component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float

math::**is_finite(x: double) : bool**()

---

is_nan

math::**is_nan(x: float) : bool**()

Returns true if x is NaN (Not a Number), checked component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float

math::**is_nan(x: double) : bool**()

---

log

math::**log(x: float4) : float4**()

Returns the natural (base-e) logarithm of x; the input must be positive; computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float4

math::**log(x: float3) : float3**()

math::**log(x: float2) : float2**()

math::**log(x: double) : double**()

math::**log(x: float) : float**()

---

log2

`math::log2(x: double) : double()`

Returns the base-2 logarithm of x; the input must be positive; computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : double

`math::log2(x: float4) : float4()`

`math::log2(x: float3) : float3()`

`math::log2(x: float2) : float2()`

`math::log2(x: float) : float()`

---

pow

`math::pow(x: double; y: double) : double()`

Returns x raised to the power of y for scalar double, float, or vector float2, float3, float4 types; domain requires x >= 0 for non-integer y values.

Arguments

- **x** : double
- **y** : double

`math::pow(x: float4; y: float4) : float4()`

`math::pow(x: float3; y: float3) : float3()`

`math::pow(x: float; y: float) : float()`

`math::pow(x: float2; y: float2) : float2()`

---

rcp

`math::rcp(x: double) : double()`

Returns the reciprocal (1/x) of a scalar float or each component of a float2, float3, or float4 vector.

Arguments

- **x** : double

`math::rcp(x: float4) : float4()`

`math::rcp(x: float3) : float3()`

`math::rcp(x: float2) : float2()`

`math::rcp(x: float) : float()`

---

safe_acos

`math::safe_acos(x: double) : double()`

Returns the arccosine of x in radians, clamping the input to the valid domain [-1, 1] to prevent NaN results from out-of-range values.

Arguments

- **x** : double

`math::safe_acos(x: float2) : float2()`

`math::safe_acos(x: float3) : float3()`

`math::safe_acos(x: float) : float()`

`math::safe_acos(x: float4) : float4()`

---

safe_asin

`math::safe_asin(x: float3) : float3()`

Returns the arcsine of x in radians, clamping the input to the valid domain [-1, 1] to prevent NaN results from out-of-range values.

Arguments

- **x** : float3

`math::safe_asin(x: float4) : float4()`

`math::safe_asin(x: double) : double()`

`math::safe_asin(x: float2) : float2()`

`math::safe_asin(x: float) : float()`

---

saturate

`math::saturate(x: float4) : float4()`

Clamps the scalar double, float, or each component of a float2, float3, float4 vector to the [0, 1] range, returning 0 for values below 0 and 1 for values above 1.

Arguments

- **x** : float4

`math::saturate(x: float3) : float3()`

`math::saturate(x: float2) : float2()`

`math::saturate(x: float) : float()`

---

sign

math::**sign(x: uint) : uint**()

Returns the sign of x component-wise: -1 for negative, 0 for zero, or 1 for positive. For unsigned types, the result is 0 or 1.

Arguments

- **x** : uint

math::**sign(x: uint2) : uint2**()

math::**sign(x: int4) : int4**()

math::**sign(x: float4) : float4**()

math::**sign(x: int64) : int64**()

math::**sign(x: double) : double**()

math::**sign(x: float3) : float3**()

math::**sign(x: uint4) : uint4**()

math::**sign(x: float) : float**()

math::**sign(x: float2) : float2**()

math::**sign(x: uint64) : uint64**()

math::**sign(x: int3) : int3**()

math::**sign(x: int) : int**()

math::**sign(x: int2) : int2**()

math::**sign(x: uint3) : uint3**()

---

sin

math::**sin(x: double) : double**()

Returns the sine of the angle x given in radians for double or float, with output in the range [-1, 1].

Arguments

- **x** : double

math::**sin(x: float4) : float4**()

math::**sin(x: float3) : float3**()

math::**sin(x: float2) : float2**()

math::**sin(x: float) : float**()

---

sincos

`math::`**`sincos`**`(`*x: float; s: float&; c: float&*`)`

Computes both the sine and cosine of the angle x in radians simultaneously, writing the results to output parameters s and c, for float or double types.

Arguments

- **x** : float
- **s** : float& implicit
- **c** : float& implicit

`math::`**`sincos`**`(`*x: double; s: double&; c: double&*`)`

---

sqrt

`math::`**`sqrt(x: double) : double`**`()`

Returns the square root of a scalar double, float, or each component of a float2, float3, or float4 vector; input must be non-negative.

Arguments

- **x** : double

`math::`**`sqrt(x: float4) : float4`**`()`

`math::`**`sqrt(x: float3) : float3`**`()`

`math::`**`sqrt(x: float2) : float2`**`()`

`math::`**`sqrt(x: float) : float`**`()`

---

tan

`math::`**`tan(x: double) : double`**`()`

Returns the tangent of the angle x given in radians for double or float; undefined at odd multiples of pi/2.

Arguments

- **x** : double

`math::`**`tan(x: float4) : float4`**`()`

`math::`**`tan(x: float2) : float2`**`()`

`math::`**`tan(x: float) : float`**`()`

`math::`**`tan(x: float3) : float3`**`()`

2.2.5 float* only

- *atan2_est (y: float; x: float) : float*
- *atan2_est (y: float4; x: float4) : float4*
- *atan2_est (y: float3; x: float3) : float3*
- *atan2_est (y: float2; x: float2) : float2*
- *atan_est (x: float2) : float2*
- *atan_est (x: float4) : float4*
- *atan_est (x: float) : float*
- *atan_est (x: float3) : float3*
- *ceili (x: float4) : int4*
- *ceili (x: float) : int*
- *ceili (x: double) : int*
- *ceili (x: float3) : int3*
- *ceili (x: float2) : int2*
- *float3x3- (x: float3x3) : float3x3*
- *float3x4- (x: float3x4) : float3x4*
- *float4x4- (x: float4x4) : float4x4*
- *floori (x: float) : int*
- *floori (x: float2) : int2*
- *floori (x: float4) : int4*
- *floori (x: double) : int*
- *floori (x: float3) : int3*
- *fract (x: float3) : float3*
- *fract (x: float4) : float4*
- *fract (x: float2) : float2*
- *fract (x: float) : float*
- *rcp_est (x: float4) : float4*
- *rcp_est (x: float2) : float2*
- *rcp_est (x: float) : float*
- *rcp_est (x: float3) : float3*
- *round (x: float4) : float4*
- *round (x: float3) : float3*
- *round (x: float2) : float2*
- *round (x: float) : float*
- *roundi (x: float4) : int4*
- *roundi (x: float2) : int2*

- *roundi (x: float3) : int3*
- *roundi (x: double) : int*
- *roundi (x: float) : int*
- *rsqrt (x: float4) : float4*
- *rsqrt (x: float2) : float2*
- *rsqrt (x: float) : float*
- *rsqrt (x: float3) : float3*
- *rsqrt_est (x: float3) : float3*
- *rsqrt_est (x: float4) : float4*
- *rsqrt_est (x: float2) : float2*
- *rsqrt_est (x: float) : float*
- *trunci (x: double) : int*
- *trunci (x: float) : int*
- *trunci (x: float2) : int2*
- *trunci (x: float4) : int4*
- *trunci (x: float3) : int3*

atan2_est

`math::atan2_est(y: float; x: float) : float()`

Returns a fast estimated arctangent of y/x in radians, using the signs of both arguments to determine the correct quadrant; trades some precision for speed.

Arguments

- **y** : float
- **x** : float

`math::atan2_est(y: float4; x: float4) : float4()`

`math::atan2_est(y: float3; x: float3) : float3()`

`math::atan2_est(y: float2; x: float2) : float2()`

---

atan_est

`math::atan_est(x: float2) : float2()`

Returns a fast estimated arctangent of x in radians, trading some precision for speed; the result approximates the range [-pi/2, pi/2].

Arguments

- **x** : float2

math::**atan_est(x: float4) : float4**()

math::**atan_est(x: float) : float**()

math::**atan_est(x: float3) : float3**()

---

ceili

math::**ceili(x: float4) : int4**()

Returns the smallest integer not less than x (rounds toward positive infinity), converting the float argument to an int result.

Arguments

- **x** : float4

math::**ceili(x: float) : int**()

math::**ceili(x: double) : int**()

math::**ceili(x: float3) : int3**()

math::**ceili(x: float2) : int2**()

---

math::**float3x3-(x: float3x3) : float3x3**()

Returns the component-wise arithmetic negation of a matrix, flipping the sign of every element; works with float3x3, float3x4, and float4x4 matrix types.

Arguments

- **x** : *float3x3* implicit

math::**float3x4-(x: float3x4) : float3x4**()

Returns the component-wise arithmetic negation of a matrix, flipping the sign of every element; works with float3x3, float3x4, and float4x4 matrix types.

Arguments

- **x** : *float3x4* implicit

math::**float4x4-(x: float4x4) : float4x4**()

Returns the component-wise arithmetic negation of a matrix, flipping the sign of every element; works with float3x3, float3x4, and float4x4 matrix types.

Arguments

- **x** : *float4x4* implicit

floori

`math::floori(x: float) : int()`

Returns the largest integer not greater than x (rounds toward negative infinity), converting the float argument to an int result.

Arguments

- **x** : float

`math::floori(x: float2) : int2()`

`math::floori(x: float4) : int4()`

`math::floori(x: double) : int()`

`math::floori(x: float3) : int3()`

---

fract

`math::fract(x: float3) : float3()`

Returns the fractional part of x (equivalent to x - floor(x)), computed component-wise for float2, float3, and float4 vector types; works with float and double scalars.

Arguments

- **x** : float3

`math::fract(x: float4) : float4()`

`math::fract(x: float2) : float2()`

`math::fract(x: float) : float()`

---

rcp_est

`math::rcp_est(x: float4) : float4()`

Returns a fast hardware estimate of the reciprocal (1/x) of a scalar float or each component of a float2, float3, or float4 vector, trading precision for speed.

Arguments

- **x** : float4

`math::rcp_est(x: float2) : float2()`

`math::rcp_est(x: float) : float()`

`math::rcp_est(x: float3) : float3()`

---

round

math::**round(x: float4) : float4**()

Rounds each component of the scalar double, float, or vector float2, float3, float4 value x to the nearest integer, with halfway cases rounded to the nearest even value.

Arguments

- **x** : float4

math::**round(x: float3) : float3**()

math::**round(x: float2) : float2**()

math::**round(x: float) : float**()

---

roundi

math::**roundi(x: float4) : int4**()

Rounds the float x to the nearest integer value and returns the result as an int.

Arguments

- **x** : float4

math::**roundi(x: float2) : int2**()

math::**roundi(x: float3) : int3**()

math::**roundi(x: double) : int**()

math::**roundi(x: float) : int**()

---

rsqrt

math::**rsqrt(x: float4) : float4**()

Returns the reciprocal square root (1/sqrt(x)) of a scalar float or each component of a float2, float3, or float4 vector.

Arguments

- **x** : float4

math::**rsqrt(x: float2) : float2**()

math::**rsqrt(x: float) : float**()

math::**rsqrt(x: float3) : float3**()

---

rsqrt_est

`math::rsqrt_est(x: float3) : float3()`

Returns a fast hardware estimate of the reciprocal square root (1/sqrt(x)) of a scalar float or each component of a float2, float3, or float4 vector, trading precision for speed.

Arguments

- **x** : float3

`math::rsqrt_est(x: float4) : float4()`

`math::rsqrt_est(x: float2) : float2()`

`math::rsqrt_est(x: float) : float()`

---

trunci

`math::trunci(x: double) : int()`

Truncates the float x toward zero to the nearest integer and returns the result as an int.

Arguments

- **x** : double

`math::trunci(x: float) : int()`

`math::trunci(x: float2) : int2()`

`math::trunci(x: float4) : int4()`

`math::trunci(x: float3) : int3()`

2.2.6 float3 only

- *cross (x: float3; y: float3) : float3*
- *distance (x: float2; y: float2) : float*
- *distance (x: float4; y: float4) : float*
- *distance (x: float3; y: float3) : float*
- *distance_sq (x: float3; y: float3) : float*
- *distance_sq (x: float2; y: float2) : float*
- *distance_sq (x: float4; y: float4) : float*
- *inv_distance (x: float3; y: float3) : float*
- *inv_distance (x: float2; y: float2) : float*
- *inv_distance (x: float4; y: float4) : float*
- *inv_distance_sq (x: float4; y: float4) : float*
- *inv_distance_sq (x: float2; y: float2) : float*

- *inv_distance_sq (x: float3; y: float3) : float*
- *reflect (v: float3; n: float3) : float3*
- *reflect (v: float2; n: float2) : float2*
- *refract (v: float2; n: float2; nint: float) : float2*
- *refract (v: float3; n: float3; nint: float) : float3*

`math::`**`cross(x: float3; y: float3) : float3`**`()`

Returns the cross product of two float3 vectors, producing a float3 vector perpendicular to both inputs with magnitude equal to the area of the parallelogram they span.

Arguments

- **x** : float3
- **y** : float3

distance

`math::`**`distance(x: float2; y: float2) : float`**`()`

Returns the Euclidean distance between two vectors as a float scalar; works with float2, float3, and float4 vector types.

Arguments

- **x** : float2
- **y** : float2

`math::`**`distance(x: float4; y: float4) : float`**`()`

`math::`**`distance(x: float3; y: float3) : float`**`()`

---

distance_sq

`math::`**`distance_sq(x: float3; y: float3) : float`**`()`

Returns the squared Euclidean distance between two vectors as a float scalar, avoiding the square root for faster distance comparisons; works with float2, float3, and float4 vector types.

Arguments

- **x** : float3
- **y** : float3

`math::`**`distance_sq(x: float2; y: float2) : float`**`()`

`math::`**`distance_sq(x: float4; y: float4) : float`**`()`

---

inv_distance

`math::inv_distance(x: float3; y: float3) : float()`

Returns the reciprocal of the Euclidean distance between two vectors (1 / distance(x, y)) as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float3
- **y** : float3

`math::inv_distance(x: float2; y: float2) : float()`

`math::inv_distance(x: float4; y: float4) : float()`

---

inv_distance_sq

`math::inv_distance_sq(x: float4; y: float4) : float()`

Returns the reciprocal of the squared Euclidean distance between two vectors (1 / distance_sq(x, y)) as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float4
- **y** : float4

`math::inv_distance_sq(x: float2; y: float2) : float()`

`math::inv_distance_sq(x: float3; y: float3) : float()`

---

reflect

`math::reflect(v: float3; n: float3) : float3()`

Computes the reflection of float2 or float3 vector v off a surface with unit normal n, returning the reflected vector as v - 2*dot(v,n)*n.

Arguments

- **v** : float3
- **n** : float3

`math::reflect(v: float2; n: float2) : float2()`

---

refract

`math::refract(v: float2; n: float2; nint: float) : float2()`

Computes the refraction direction of vector v through a surface with unit normal n using Snell's law with index of refraction ratio nint. Returns a zero vector if total internal reflection occurs.

Arguments

- **v** : float2
- **n** : float2
- **nint** : float

`math::refract(v: float3; n: float3; nint: float) : float3()`

2.2.7 float2, float3, float4

- *dot (x: float3; y: float3) : float*
- *dot (x: float2; y: float2) : float*
- *dot (x: float4; y: float4) : float*
- *fast_normalize (x: float2) : float2*
- *fast_normalize (x: float4) : float4*
- *fast_normalize (x: float3) : float3*
- *inv_length (x: float4) : float*
- *inv_length (x: float3) : float*
- *inv_length (x: float2) : float*
- *inv_length_sq (x: float4) : float*
- *inv_length_sq (x: float2) : float*
- *inv_length_sq (x: float3) : float*
- *length (x: float3) : float*
- *length (x: float2) : float*
- *length (x: float4) : float*
- *length_sq (x: float3) : float*
- *length_sq (x: float2) : float*
- *length_sq (x: float4) : float*
- *normalize (x: float2) : float2*
- *normalize (x: float3) : float3*
- *normalize (x: float4) : float4*

dot

`math::dot(x: float3; y: float3) : float()`

Returns the dot product (scalar product) of two vectors as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float3
- **y** : float3

`math::dot(x: float2; y: float2) : float()`

`math::dot(x: float4; y: float4) : float()`

---

fast_normalize

`math::fast_normalize(x: float2) : float2()`

Returns a unit-length vector in the same direction as x using a fast approximation; does not check for zero-length input; works with float2, float3, and float4 vector types.

Arguments

- **x** : float2

`math::fast_normalize(x: float4) : float4()`

`math::fast_normalize(x: float3) : float3()`

---

inv_length

`math::inv_length(x: float4) : float()`

Returns the reciprocal of the length of the vector (1 / length(x)) as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float4

`math::inv_length(x: float3) : float()`

`math::inv_length(x: float2) : float()`

---

inv_length_sq

math::**inv_length_sq(x: float4) : float**()

Returns the reciprocal of the squared length of the vector (1 / length_sq(x)) as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float4

math::**inv_length_sq(x: float2) : float**()

math::**inv_length_sq(x: float3) : float**()

---

length

math::**length(x: float3) : float**()

Returns the Euclidean length (magnitude) of the vector as a float; works with float2, float3, and float4 vector types.

Arguments

- **x** : float3

math::**length(x: float2) : float**()

math::**length(x: float4) : float**()

---

length_sq

math::**length_sq(x: float3) : float**()

Returns the squared Euclidean length of the vector as a float, equivalent to dot(x, x) and avoiding the square root for faster magnitude comparisons; works with float2, float3, and float4 vector types.

Arguments

- **x** : float3

math::**length_sq(x: float2) : float**()

math::**length_sq(x: float4) : float**()

---

normalize

`math::normalize(x: float2) : float2()`

Returns a unit-length vector with the same direction as the input float2, float3, or float4 vector; behavior is undefined if the input vector has zero length.

Arguments

- **x** : float2

`math::normalize(x: float3) : float3()`

`math::normalize(x: float4) : float4()`

2.2.8 Noise functions

- *uint32_hash (seed: uint) : uint*
- *uint_noise_1D (position: int; seed: uint) : uint*
- *uint_noise_2D (position: int2; seed: uint) : uint*
- *uint_noise_3D (position: int3; seed: uint) : uint*

`math::uint32_hash(seed: uint) : uint()`

Returns a well-distributed uint hash of the input uint seed using an improved integer hash function suitable for hash tables and procedural generation.

Arguments

- **seed** : uint

`math::uint_noise_1D(position: int; seed: uint) : uint()`

Generates a deterministic uint hash value from a 1D integer position and a uint seed, suitable for repeatable procedural noise.

Arguments

- **position** : int
- **seed** : uint

`math::uint_noise_2D(position: int2; seed: uint) : uint()`

Generates a deterministic uint hash value from 2D integer coordinates (x, y) and a uint seed, suitable for repeatable procedural noise.

Arguments

- **position** : int2
- **seed** : uint

`math::uint_noise_3D(position: int3; seed: uint) : uint()`

Generates a deterministic uint hash value from 3D integer coordinates (x, y, z) and a uint seed, suitable for repeatable procedural noise.

Arguments

- **position** : int3

- **seed** : uint

2.2.9 lerp/mad/clamp

- *clamp (t: float; a: float; b: float) : float*
- *clamp (t: int64; a: int64; b: int64) : int64*
- *clamp (t: int3; a: int3; b: int3) : int3*
- *clamp (t: uint3; a: uint3; b: uint3) : uint3*
- *clamp (t: int; a: int; b: int) : int*
- *clamp (t: uint64; a: uint64; b: uint64) : uint64*
- *clamp (t: int4; a: int4; b: int4) : int4*
- *clamp (t: int2; a: int2; b: int2) : int2*
- *clamp (t: float4; a: float4; b: float4) : float4*
- *clamp (t: float2; a: float2; b: float2) : float2*
- *clamp (t: uint4; a: uint4; b: uint4) : uint4*
- *clamp (t: float3; a: float3; b: float3) : float3*
- *clamp (t: double; a: double; b: double) : double*
- *clamp (t: uint; a: uint; b: uint) : uint*
- *clamp (t: uint2; a: uint2; b: uint2) : uint2*
- *lerp (a: double; b: double; t: double) : double*
- *lerp (a: float4; b: float4; t: float4) : float4*
- *lerp (a: float3; b: float3; t: float3) : float3*
- *lerp (a: float2; b: float2; t: float2) : float2*
- *lerp (a: float; b: float; t: float) : float*
- *lerp (a: float3; b: float3; t: float) : float3*
- *lerp (a: float4; b: float4; t: float) : float4*
- *lerp (a: float2; b: float2; t: float) : float2*
- *mad (a: uint4; b: uint4; c: uint4) : uint4*
- *mad (a: uint3; b: uint3; c: uint3) : uint3*
- *mad (a: uint2; b: uint; c: uint2) : uint2*
- *mad (a: uint; b: uint; c: uint) : uint*
- *mad (a: int3; b: int; c: int3) : int3*
- *mad (a: int2; b: int; c: int2) : int2*
- *mad (a: int4; b: int; c: int4) : int4*
- *mad (a: uint2; b: uint2; c: uint2) : uint2*
- *mad (a: uint3; b: uint; c: uint3) : uint3*
- *mad (a: int2; b: int2; c: int2) : int2*

- *mad (a: float4; b: float; c: float4) : float4*
- *mad (a: float3; b: float; c: float3) : float3*
- *mad (a: int; b: int; c: int) : int*
- *mad (a: int3; b: int3; c: int3) : int3*
- *mad (a: float4; b: float4; c: float4) : float4*
- *mad (a: double; b: double; c: double) : double*
- *mad (a: float; b: float; c: float) : float*
- *mad (a: float2; b: float2; c: float2) : float2*
- *mad (a: float3; b: float3; c: float3) : float3*
- *mad (a: float2; b: float; c: float2) : float2*
- *mad (a: int4; b: int4; c: int4) : int4*
- *mad (a: uint4; b: uint; c: uint4) : uint4*

clamp

`math::clamp(t: float; a: float; b: float) : float()`

Returns the value t clamped to the inclusive range [a, b], equivalent to min(max(t, a), b); works with float, double, float2, float3, float4, int, int64, uint, and uint64 types.

Arguments

- **t** : float
- **a** : float
- **b** : float

`math::clamp(t: int64; a: int64; b: int64) : int64()`

`math::clamp(t: int3; a: int3; b: int3) : int3()`

`math::clamp(t: uint3; a: uint3; b: uint3) : uint3()`

`math::clamp(t: int; a: int; b: int) : int()`

`math::clamp(t: uint64; a: uint64; b: uint64) : uint64()`

`math::clamp(t: int4; a: int4; b: int4) : int4()`

`math::clamp(t: int2; a: int2; b: int2) : int2()`

`math::clamp(t: float4; a: float4; b: float4) : float4()`

`math::clamp(t: float2; a: float2; b: float2) : float2()`

`math::clamp(t: uint4; a: uint4; b: uint4) : uint4()`

`math::clamp(t: float3; a: float3; b: float3) : float3()`

`math::clamp(t: double; a: double; b: double) : double()`

`math::clamp(t: uint; a: uint; b: uint) : uint()`

math::**clamp(t: uint2; a: uint2; b: uint2) : uint2**()

---

lerp

math::**lerp(a: double; b: double; t: double) : double**()

Performs linear interpolation between a and b using the factor t, returning a + (b - a) * t; when t is 0 the result is a, when t is 1 the result is b; works component-wise with float, double, float2, float3, and float4 types.

Arguments

- **a** : double
- **b** : double
- **t** : double

math::**lerp(a: float4; b: float4; t: float4) : float4**()

math::**lerp(a: float3; b: float3; t: float3) : float3**()

math::**lerp(a: float2; b: float2; t: float2) : float2**()

math::**lerp(a: float; b: float; t: float) : float**()

math::**lerp(a: float3; b: float3; t: float) : float3**()

math::**lerp(a: float4; b: float4; t: float) : float4**()

math::**lerp(a: float2; b: float2; t: float) : float2**()

---

mad

math::**mad(a: uint4; b: uint4; c: uint4) : uint4**()

Computes the fused multiply-add operation $a * b + c$.

Arguments

- **a** : uint4
- **b** : uint4
- **c** : uint4

math::**mad(a: uint3; b: uint3; c: uint3) : uint3**()

math::**mad(a: uint2; b: uint; c: uint2) : uint2**()

math::**mad(a: uint; b: uint; c: uint) : uint**()

math::**mad(a: int3; b: int; c: int3) : int3**()

math::**mad(a: int2; b: int; c: int2) : int2**()

math::**mad(a: int4; b: int; c: int4) : int4**()

math::**mad(a: uint2; b: uint2; c: uint2) : uint2**()

math::**mad(a: uint3; b: uint; c: uint3) : uint3**()

math::**mad(a: int2; b: int2; c: int2) : int2**()

math::**mad(a: float4; b: float; c: float4) : float4**()

math::**mad(a: float3; b: float; c: float3) : float3**()

math::**mad(a: int; b: int; c: int) : int**()

math::**mad(a: int3; b: int3; c: int3) : int3**()

math::**mad(a: float4; b: float4; c: float4) : float4**()

math::**mad(a: double; b: double; c: double) : double**()

math::**mad(a: float; b: float; c: float) : float**()

math::**mad(a: float2; b: float2; c: float2) : float2**()

math::**mad(a: float3; b: float3; c: float3) : float3**()

math::**mad(a: float2; b: float; c: float2) : float2**()

math::**mad(a: int4; b: int4; c: int4) : int4**()

math::**mad(a: uint4; b: uint; c: uint4) : uint4**()

2.2.10 Matrix element access

- *float3x3 const implicit ==const.[] (m: float3x3 const implicit ==const; i: int) : float3*
- *float3x3 const implicit ==const.[] (m: float3x3 const implicit ==const; i: uint) : float3*
- *float3x3 implicit ==const.[] (m: float3x3 implicit ==const; i: int) : float3&*
- *float3x3 implicit ==const.[] (m: float3x3 implicit ==const; i: uint) : float3&*
- *float3x4 const implicit ==const.[] (m: float3x4 const implicit ==const; i: uint) : float3*
- *float3x4 const implicit ==const.[] (m: float3x4 const implicit ==const; i: int) : float3*
- *float3x4 implicit ==const.[] (m: float3x4 implicit ==const; i: uint) : float3&*
- *float3x4 implicit ==const.[] (m: float3x4 implicit ==const; i: int) : float3&*
- *float4x4 const implicit ==const.[] (m: float4x4 const implicit ==const; i: uint) : float4*
- *float4x4 const implicit ==const.[] (m: float4x4 const implicit ==const; i: int) : float4*
- *float4x4 implicit ==const.[] (m: float4x4 implicit ==const; i: int) : float4&*
- *float4x4 implicit ==const.[] (m: float4x4 implicit ==const; i: uint) : float4&*

float3x3 const implicit ==const.[]

`float3x3 const implicit ==const.[](m: float3x3 const implicit ==const; i: int) : float3()`

Returns a copy of the row vector at index *i* from a constant float3x3 matrix.

Arguments

- **m** : *float3x3* implicit!
- **i** : int

`float3x3 const implicit ==const.[](m: float3x3 const implicit ==const; i: uint) : float3()`

---

float3x3 implicit ==const.[]

`float3x3 implicit ==const.[](m: float3x3 implicit ==const; i: int) : float3&()`

Returns a reference to the row vector at index *i* from a float3x3 matrix.

Arguments

- **m** : *float3x3* implicit!
- **i** : int

`float3x3 implicit ==const.[](m: float3x3 implicit ==const; i: uint) : float3&()`

---

float3x4 const implicit ==const.[]

`float3x4 const implicit ==const.[](m: float3x4 const implicit ==const; i: uint) : float3()`

Returns a copy of the row vector at index *i* from a constant float3x4 matrix.

Arguments

- **m** : *float3x4* implicit!
- **i** : uint

`float3x4 const implicit ==const.[](m: float3x4 const implicit ==const; i: int) : float3()`

---

float3x4 implicit ==const.[]

`float3x4 implicit ==const.[](m: float3x4 implicit ==const; i: uint) : float3&()`

Returns a reference to the row vector at index *i* from a float3x4 matrix.

Arguments

- **m** : *float3x4* implicit!
- **i** : uint

`float3x4 implicit ==const.[](m: float3x4 implicit ==const; i: int) : float3&()`

---

float4x4 const implicit ==const.[]

`float4x4 const implicit ==const.[](m: float4x4 const implicit ==const; i: uint) : float4()`

Returns a copy of the row vector at index *i* from a constant float4x4 matrix.

Arguments

- **m** : *float4x4* implicit!
- **i** : uint

`float4x4 const implicit ==const.[](m: float4x4 const implicit ==const; i: int) : float4()`

float4x4 implicit ==const.[]

`float4x4 implicit ==const.[](m: float4x4 implicit ==const; i: int) : float4&()`

Returns a reference to the row vector at index *i* from a float4x4 matrix.

Arguments

- **m** : *float4x4* implicit!
- **i** : int

`float4x4 implicit ==const.[](m: float4x4 implicit ==const; i: uint) : float4&()`

2.2.11 Matrix operations

- *float3x3!= (x: float3x3; y: float3x3) : bool*
- *float3x3* (x: float3x3; y: float3x3) : float3x3*
- *float3x3* (x: float3x3; y: float3) : float3*
- *float3x3== (x: float3x3; y: float3x3) : bool*
- *float3x4!= (x: float3x4; y: float3x4) : bool*
- *float3x4* (x: float3x4; y: float3x4) : float3x4*
- *float3x4* (x: float3x4; y: float3) : float3*
- *float3x4== (x: float3x4; y: float3x4) : bool*
- *float4x4!= (x: float4x4; y: float4x4) : bool*
- *float4x4* (x: float4x4; y: float4) : float4*
- *float4x4* (x: float4x4; y: float4x4) : float4x4*
- *float4x4== (x: float4x4; y: float4x4) : bool*

`math::float3x3!=(x: float3x3; y: float3x3) : bool()`

Returns true if two float3x3 matrices are not equal, comparing all elements component-wise.

Arguments

- **x** : *float3x3* implicit
- **y** : *float3x3* implicit

float3x3*

`math::float3x3*(x: float3x3; y: float3x3) : float3x3()`

Transforms a float3 vector by a 3x3 matrix.

Arguments

- **x** : *float3x3* implicit
- **y** : *float3x3* implicit

`math::float3x3*(x: float3x3; y: float3) : float3()`

`math::float3x3==(x: float3x3; y: float3x3) : bool()`

Returns true if two float3x3 matrices are exactly equal, comparing all elements component-wise.

Arguments

- **x** : *float3x3* implicit
- **y** : *float3x3* implicit

`math::float3x4!=(x: float3x4; y: float3x4) : bool()`

Returns true if two float3x4 matrices are not equal, comparing all elements component-wise.

Arguments

- **x** : *float3x4* implicit
- **y** : *float3x4* implicit

float3x4*

`math::float3x4*(x: float3x4; y: float3x4) : float3x4()`

Transforms a float3 vector by a 3x3 matrix.

Arguments

- **x** : *float3x4* implicit
- **y** : *float3x4* implicit

`math::float3x4*(x: float3x4; y: float3) : float3()`

`math::float3x4==(x: float3x4; y: float3x4) : bool()`

Returns true if two float3x4 matrices are exactly equal, comparing all elements component-wise.

Arguments

- **x** : *float3x4* implicit
- **y** : *float3x4* implicit

`math::`**`float4x4!=(x: float4x4; y: float4x4) : bool`**`()`

Returns true if two float4x4 matrices are not equal, comparing all elements component-wise.

Arguments

- **x** : *float4x4* implicit
- **y** : *float4x4* implicit

float4x4*

`math::`**`float4x4*(x: float4x4; y: float4) : float4`**`()`

Transforms a float4 vector by a 4x4 matrix.

Arguments

- **x** : *float4x4* implicit
- **y** : float4

`math::`**`float4x4*(x: float4x4; y: float4x4) : float4x4`**`()`

`math::`**`float4x4==(x: float4x4; y: float4x4) : bool`**`()`

Returns true if two float4x4 matrices are exactly equal, comparing all elements component-wise.

Arguments

- **x** : *float4x4* implicit
- **y** : *float4x4* implicit

2.2.12 Matrix initializers

- *float3x3 (arg0: float3x4) : float3x3*
- *float3x3 () : float3x3*
- *float3x3 (arg0: float4x4) : float3x3*
- *float3x4 (arg0: float4x4) : float3x4*
- *float3x4 () : float3x4*
- *float4x4 (arg0: float3x4) : float4x4*
- *float4x4 () : float4x4*
- *identity3x3 () : float3x3*
- *identity3x4 () : float3x4*
- *identity4x4 () : float4x4*

float3x3

`math::float3x3(arg0: float3x4) : float3x3()`

Extracts the upper-left 3x3 rotation part from a float3x4 transformation matrix, returning it as a float3x3.

Arguments

- **arg0** : *float3x4* implicit

`math::float3x3() : float3x3()`

`math::float3x3(arg0: float4x4) : float3x3()`

---

float3x4

`math::float3x4(arg0: float4x4) : float3x4()`

Constructs a float3x4 transformation matrix from a float3x3 rotation matrix, with the translation component set to zero.

Arguments

- **arg0** : *float4x4* implicit

`math::float3x4() : float3x4()`

---

float4x4

`math::float4x4(arg0: float3x4) : float4x4()`

Converts a float3x4 transformation matrix to a float4x4 matrix, filling the fourth row with [0, 0, 0, 1].

Arguments

- **arg0** : *float3x4* implicit

`math::float4x4() : float4x4()`

`math::identity3x3() : float3x3()`

Returns a float3x3 identity matrix with ones on the diagonal and zeros elsewhere.

`math::identity3x4() : float3x4()`

Returns a float3x4 identity transformation matrix with the rotation part set to identity and the translation set to zero.

`math::identity4x4() : float4x4()`

Returns a float4x4 identity matrix with ones on the diagonal and zeros elsewhere.

2.2.13 Matrix manipulation

- *compose (pos: float3; rot: float4; scale: float3) : float4x4*
- *decompose (mat: float4x4; pos: float3&; rot: float4&; scale: float3&)*
- *determinant (x: float3x4) : float*
- *determinant (x: float4x4) : float*
- *determinant (x: float3x3) : float*
- *identity (x: float3x4)*
- *identity (x: float4x4)*
- *identity (x: float3x3)*
- *inverse (x: float3x4) : float3x4*
- *inverse (m: float4x4) : float4x4*
- *look_at (eye: float3; at: float3; up: float3) : float4x4*
- *orthonormal_inverse (m: float3x3) : float3x3*
- *orthonormal_inverse (m: float3x4) : float3x4*
- *persp_forward (wk: float; hk: float; zn: float; zf: float) : float4x4*
- *persp_reverse (wk: float; hk: float; zn: float; zf: float) : float4x4*
- *rotate (x: float3x4; y: float3) : float3*
- *translation (xyz: float3) : float4x4*
- *transpose (x: float4x4) : float4x4*

`math::`**`compose(pos: float3; rot: float4; scale: float3) : float4x4`**`()`

Constructs a float4x4 transformation matrix from a float3 translation position, a float4 quaternion rotation, and a float3 scale.

Arguments

- **pos** : float3
- **rot** : float4
- **scale** : float3

`math::`**`decompose`**`(`*mat: float4x4; pos: float3&; rot: float4&; scale: float3&*`)`

Decomposes a float4x4 transformation matrix into its float3 translation position, float4 quaternion rotation, and float3 scale components., writing the results into the output arguments rot and pos.

Arguments

- **mat** : *float4x4* implicit
- **pos** : float3& implicit
- **rot** : float4& implicit
- **scale** : float3& implicit

determinant

`math::determinant(x: float3x4) : float()`

Returns the determinant of a float3x4 matrix as a float scalar; a zero determinant indicates the matrix is singular and non-invertible.

Arguments

- **x** : *float3x4* implicit

`math::determinant(x: float4x4) : float()`

`math::determinant(x: float3x3) : float()`

---

identity

`math::identity(x: float3x4)`

Sets the given float3x4 matrix to the identity transformation (rotation part is the identity matrix, translation is zero) and returns it.

Arguments

- **x** : *float3x4* implicit

`math::identity(x: float4x4)`

`math::identity(x: float3x3)`

---

inverse

`math::inverse(x: float3x4) : float3x4()`

Returns the inverse of a matrix, such that multiplying the original by its inverse yields the identity; works with float3x4 and float4x4 matrix types.

Arguments

- **x** : *float3x4* implicit

`math::inverse(m: float4x4) : float4x4()`

`math::look_at(eye: float3; at: float3; up: float3) : float4x4()`

Constructs a float4x4 look-at view transformation matrix from eye position, target position, and up vector. from an eye position, a target point to look at, and an up direction vector.

Arguments

- **eye** : float3
- **at** : float3
- **up** : float3

orthonormal_inverse

`math::orthonormal_inverse(m: float3x3) : float3x3()`

Returns the inverse of a float3x3 orthonormal matrix (each axis is unit length and mutually perpendicular), computed more efficiently than a general matrix inverse.

Arguments

- **m** : *float3x3* implicit

`math::orthonormal_inverse(m: float3x4) : float3x4()`

`math::persp_forward(wk: float; hk: float; zn: float; zf: float) : float4x4()`

Returns a forward (standard) perspective projection float4x4 matrix constructed from horizontal scale wk, vertical scale hk, near plane zn, and far plane zf.

Arguments

- **wk** : float
- **hk** : float
- **zn** : float
- **zf** : float

`math::persp_reverse(wk: float; hk: float; zn: float; zf: float) : float4x4()`

Returns a reverse-depth perspective projection float4x4 matrix constructed from horizontal scale wk, vertical scale hk, near plane zn, and far plane zf, mapping the far plane to 0 and the near plane to 1.

Arguments

- **wk** : float
- **hk** : float
- **zn** : float
- **zf** : float

`math::rotate(x: float3x4; y: float3) : float3()`

Rotates a float3 vector v by the 3x3 rotation part of the float3x4 matrix m, ignoring the translation component.

Arguments

- **x** : *float3x4* implicit
- **y** : float3

`math::translation(xyz: float3) : float4x4()`

Constructs a float4x4 matrix representing a pure translation by the given float3 offset. by the float3 offset xyz, with the rotation part set to identity.

Arguments

- **xyz** : float3

`math::transpose(x: float4x4) : float4x4()`

Returns the transpose of a float3x3 or float4x4 matrix, swapping rows and columns.

Arguments

- **x** : *float4x4* implicit

2.2.14 Quaternion operations

- *euler_from_quat (angles: float4) : float3*
- *quat (m: float4x4) : float4*
- *quat (m: float3x3) : float4*
- *quat (m: float3x4) : float4*
- *quat_conjugate (q: float4) : float4*
- *quat_from_euler (x: float; y: float; z: float) : float4*
- *quat_from_euler (angles: float3) : float4*
- *quat_from_unit_arc (v0: float3; v1: float3) : float4*
- *quat_from_unit_vec_ang (v: float3; ang: float) : float4*
- *quat_mul (q1: float4; q2: float4) : float4*
- *quat_mul_vec (q: float4; v: float3) : float3*
- *quat_slerp (t: float; a: float4; b: float4) : float4*

`math::euler_from_quat(angles: float4) : float3()`

Converts a float4 quaternion to Euler angles, returning a float3 representing rotation around the x, y, and z axes in radians.

Arguments

- **angles** : float4

quat

`math::quat(m: float4x4) : float4()`

Extracts the rotation part of a float3x4 matrix and returns it as a float4 quaternion in (x, y, z, w) format.

Arguments

- **m** : *float4x4* implicit

`math::quat(m: float3x3) : float4()`

`math::quat(m: float3x4) : float4()`

math::**quat_conjugate(q: float4) : float4**()

Returns the conjugate of the float4 quaternion q by negating its xyz components, which for unit quaternions equals the inverse rotation.

Arguments

- **q** : float4

quat_from_euler

math::**quat_from_euler(x: float; y: float; z: float) : float4**()

Creates a float4 quaternion from a float3 of Euler angles (pitch, yaw, roll) given in radians.

Arguments

- **x** : float
- **y** : float
- **z** : float

math::**quat_from_euler(angles: float3) : float4**()

---

math::**quat_from_unit_arc(v0: float3; v1: float3) : float4**()

Creates a float4 quaternion representing the shortest rotation arc from unit-length float3 vector v0 to unit-length float3 vector v1; both input vectors must be normalized.

Arguments

- **v0** : float3
- **v1** : float3

math::**quat_from_unit_vec_ang(v: float3; ang: float) : float4**()

Creates a float4 quaternion representing a rotation of ang radians around the unit-length float3 axis vector v.

Arguments

- **v** : float3
- **ang** : float

math::**quat_mul(q1: float4; q2: float4) : float4**()

Returns the float4 quaternion product of q1 and q2, representing the combined rotation of q2 followed by q1.

Arguments

- **q1** : float4
- **q2** : float4

math::**quat_mul_vec(q: float4; v: float3) : float3**()

Rotates a float3 vector v by the float4 quaternion q and returns the resulting float3 vector.

Arguments

- **q** : float4

- **v** : float3

`math::`**`quat_slerp(t: float; a: float4; b: float4) : float4`**`()`

Performs spherical linear interpolation between float4 quaternions a and b by factor t in the range [0,1], returning a smoothly interpolated float4 quaternion.

Arguments

- **t** : float
- **a** : float4
- **b** : float4

2.2.15 Packing and unpacking

- *pack_float_to_byte (x: float4) : uint*
- *unpack_byte_to_float (x: uint) : float4*

`math::`**`pack_float_to_byte(x: float4) : uint`**`()`

Packs a float4 vector into a single uint by converting each component to an 8-bit byte value, mapping the XYZW components to the four bytes of the result.

Arguments

- **x** : float4

`math::`**`unpack_byte_to_float(x: uint) : float4`**`()`

Unpacks the four bytes of a uint into a float4 vector, converting each 8-bit byte value to its corresponding floating-point component.

Arguments

- **x** : uint

2.3 Math bit helpers

The MATH_BITS module provides bit manipulation functions for floating point numbers, including type punning between integer and float representations, and efficient integer math operations like `int_bits_to_float` and `float_bits_to_int`.

All functions and symbols are in “math_bits” module, use require to get access to it.

```
require daslib/math_bits
```

Example:

```
require daslib/math_bits

    [export]
    def main() {
        let f = uint_bits_to_float(0x3F800000u)
        print("uint_bits_to_float(0x3F800000) = {f}\n")
        let back = float_bits_to_uint(1.0)
```

(continues on next page)

(continued from previous page)

```
        print("float_bits_to_uint(1.0) = {back}\n")
    }
    // output:
    // uint_bits_to_float(0x3F800000) = 1
    // float_bits_to_uint(1.0) = 0x3f800000
```

2.3.1 float in int,uint

- *int_bits_to_float (x: int2) : float2*
- *int_bits_to_float (x: int) : float*
- *int_bits_to_float (x: int4) : float4*
- *int_bits_to_float (x: int3) : float3*
- *uint_bits_to_float (x: uint3) : float3*
- *uint_bits_to_float (x: uint2) : float2*
- *uint_bits_to_float (x: uint) : float*
- *uint_bits_to_float (x: uint4) : float4*

int_bits_to_float

`math_bits::`**`int_bits_to_float(x: int2) : float2`**`()`

bit representation of x is interpreted as a float

Arguments

- **x** : int2

`math_bits::`**`int_bits_to_float(x: int) : float`**`()`

`math_bits::`**`int_bits_to_float(x: int4) : float4`**`()`

`math_bits::`**`int_bits_to_float(x: int3) : float3`**`()`

---

uint_bits_to_float

`math_bits::`**`uint_bits_to_float(x: uint3) : float3`**`()`

bit representation of x is interpreted as a float3

Arguments

- **x** : uint3

`math_bits::`**`uint_bits_to_float(x: uint2) : float2`**`()`

`math_bits::`**`uint_bits_to_float(x: uint) : float`**`()`

`math_bits::`**`uint_bits_to_float(x: uint4) : float4`**`()`

2.3.2 int,uint in float

- *float_bits_to_int (x: float2) : int2*
- *float_bits_to_int (x: float) : int*
- *float_bits_to_int (x: float4) : int4*
- *float_bits_to_int (x: float3) : int3*
- *float_bits_to_uint (x: float3) : uint3*
- *float_bits_to_uint (x: float2) : uint2*
- *float_bits_to_uint (x: float) : uint*
- *float_bits_to_uint (x: float4) : uint4*

float_bits_to_int

`math_bits::`**`float_bits_to_int(x: float2) : int2`**`()`

bit representation of x is interpreted as an int2

Arguments

- **x** : float2

`math_bits::`**`float_bits_to_int(x: float) : int`**`()`

`math_bits::`**`float_bits_to_int(x: float4) : int4`**`()`

`math_bits::`**`float_bits_to_int(x: float3) : int3`**`()`

float_bits_to_uint

`math_bits::`**`float_bits_to_uint(x: float3) : uint3`**`()`

bit representation of x is interpreted as a uint3

Arguments

- **x** : float3

`math_bits::`**`float_bits_to_uint(x: float2) : uint2`**`()`

`math_bits::`**`float_bits_to_uint(x: float) : uint`**`()`

`math_bits::`**`float_bits_to_uint(x: float4) : uint4`**`()`

2.3.3 int64,uint64 in double

- *double_bits_to_int64 (x: double) : int64*
- *double_bits_to_uint64 (x: double) : uint64*
- *int64_bits_to_double (x: int64) : double*
- *uint64_bits_to_double (x: uint64) : double*

`math_bits::`**`double_bits_to_int64(x: double) : int64`**`()`

bit representation of x is interpreted as a int64

Arguments

- **x** : double

`math_bits::`**`double_bits_to_uint64(x: double) : uint64`**`()`

bit representation of x is interpreted as a uint64

Arguments

- **x** : double

`math_bits::`**`int64_bits_to_double(x: int64) : double`**`()`

bit representation of x is interpreted as a double

Arguments

- **x** : int64

`math_bits::`**`uint64_bits_to_double(x: uint64) : double`**`()`

bit representation of x is interpreted as a double

Arguments

- **x** : uint64

2.3.4 bit-cast vec4f

- *cast_to_int16 (data: float4) : int16*
- *cast_to_int32 (data: float4) : int*
- *cast_to_int64 (data: float4) : int64*
- *cast_to_int8 (data: float4) : int8*
- *cast_to_pointer (data: float4) : void?*
- *cast_to_string (data: float4) : string*
- *cast_to_vec4f (x: int64) : float4*
- *cast_to_vec4f (x: bool) : float4*

`math_bits::`**`cast_to_int16(data: float4) : int16`**`()`

return an int16 which was bit-cast from x

Arguments

- **data** : float4

`math_bits::`**`cast_to_int32(data: float4) : int`**`()`

return an int32 which was bit-cast from x

Arguments

- **data** : float4

`math_bits::`**`cast_to_int64(data: float4) : int64`**`()`

return an int64 which was bit-cast from x

Arguments

- **data** : float4

`math_bits::`**`cast_to_int8(data: float4) : int8`**`()`

return an int8 which was bit-cast from x

Arguments

- **data** : float4

`math_bits::`**`cast_to_pointer(data: float4) : void?`**`()`

return a pointer which was bit-cast from x

Arguments

- **data** : float4

`math_bits::`**`cast_to_string(data: float4) : string`**`()`

return a string which pointer was bit-cast from x

Arguments

- **data** : float4

cast_to_vec4f

`math_bits::`**`cast_to_vec4f(x: int64) : float4`**`()`

return a float4 which stores bit-cast version of x

Arguments

- **x** : int64

`math_bits::`**`cast_to_vec4f(x: bool) : float4`**`()`

2.4 Boost package for math

The MATH_BOOST module adds geometric types (`AABB`, `AABR`, `Ray`), angle conversion (`degrees`, `radians`), intersection tests, color space conversion (`linear_to_SRGB`, `RGBA_TO_UCOLOR`), and view/projection matrix construction (`look_at_lh`, `perspective_rh`).

All functions and symbols are in "math_boost" module, use require to get access to it.

```
require daslib/math_boost
```

Example:

```
require daslib/math_boost

    [export]
    def main() {
        print("degrees(PI) = {degrees(PI)}\n")
        print("radians(180) = {radians(180.0)}\n")
        var box = AABB(min = float3(0), max = float3(10))
        print("box = ({box.min}) - ({box.max})\n")
    }
    // output:
    // degrees(PI) = 180
    // radians(180) = 3.1415927
    // box = (0,0,0) - (10,10,10)
```

2.4.1 Structures

`math_boost::`**`AABR`**

axis aligned bounding rectangle

Fields

- **min** : float2 - min coordinates
- **max** : float2 - max coordinates

`math_boost::`**`AABB`**

axis aligned bounding box

Fields

- **min** : float3 - min coordinates
- **max** : float3 - max coordinates

`math_boost::`**`Ray`**

ray (direction and origin)

Fields

- **dir** : float3 - direction
- **origin** : float3 - origin

2.4.2 Angle conversions

- *degrees (f: float) : float*
- *radians (f: float) : float*

`math_boost::`**`degrees(f: float) : float`**`()`

convert radians to degrees

Arguments

- **f** : float

`math_boost::radians(f: float) : float()`

convert degrees to radians

Arguments

- **f** : float

2.4.3 Intersections

- *is_intersecting (a: AABB; b: AABB) : bool*
- *is_intersecting (a: AABR; b: AABR) : bool*
- *is_intersecting (ray: Ray; aabb: AABB; Tmin: float = 0f; Tmax: float = FLT_MAX) : bool*

is_intersecting

`math_boost::is_intersecting(a: AABB; b: AABB) : bool()`

returns true if inputs intersect

Arguments

- **a** : *AABB*
- **b** : *AABB*

`math_boost::is_intersecting(a: AABR; b: AABR) : bool()`

`math_boost::is_intersecting(ray: Ray; aabb: AABB; Tmin: float = 0f; Tmax: float = FLT_MAX) : bool()`

2.4.4 Matrices

- *look_at_lh (Eye: float3; At: float3; Up: float3) : float4x4*
- *look_at_rh (Eye: float3; At: float3; Up: float3) : float4x4*
- *ortho_rh (left: float; right: float; bottom: float; top: float; zNear: float; zFar: float) : float4x4*
- *perspective_lh (fovy: float; aspect: float; zn: float; zf: float) : float4x4*
- *perspective_rh (fovy: float; aspect: float; zn: float; zf: float) : float4x4*
- *perspective_rh_opengl (fovy: float; aspect: float; zn: float; zf: float) : float4x4*
- *planar_shadow (Light: float4; Plane: float4) : float4x4*

`math_boost::look_at_lh(Eye: float3; At: float3; Up: float3) : float4x4()`

left-handed (z forward) look at matrix with origin at *Eye* and target at *At*, and up vector *Up*.

Arguments

- **Eye** : float3
- **At** : float3
- **Up** : float3

math_boost::**look_at_rh(Eye: float3; At: float3; Up: float3) : float4x4**()

right-handed (z towards viewer) look at matrix with origin at *Eye* and target at *At*, and up vector *Up*.

Arguments

- **Eye** : float3
- **At** : float3
- **Up** : float3

math_boost::**ortho_rh(left: float; right: float; bottom: float; top: float; zNear: float; zFar: float) :**

right handed (z towards viewer) orthographic (parallel) projection matrix

Arguments

- **left** : float
- **right** : float
- **bottom** : float
- **top** : float
- **zNear** : float
- **zFar** : float

math_boost::**perspective_lh(fovy: float; aspect: float; zn: float; zf: float) : float4x4**()

left-handed (z forward) perspective matrix

Arguments

- **fovy** : float
- **aspect** : float
- **zn** : float
- **zf** : float

math_boost::**perspective_rh(fovy: float; aspect: float; zn: float; zf: float) : float4x4**()

right-handed (z toward viewer) perspective matrix

Arguments

- **fovy** : float
- **aspect** : float
- **zn** : float
- **zf** : float

math_boost::**perspective_rh_opengl(fovy: float; aspect: float; zn: float; zf: float) : float4x4**()

right-handed (z toward viewer) opengl (z in [-1..1]) perspective matrix

Arguments

- **fovy** : float
- **aspect** : float
- **zn** : float

- **zf** : float

math_boost::**planar_shadow(Light: float4; Plane: float4) : float4x4**()

planar shadow projection matrix, i.e. all light shadows to be projected on a plane

Arguments

- **Light** : float4
- **Plane** : float4

2.4.5 Plane

- *plane_dot (Plane: float4; Vec: float4) : float*
- *plane_from_point_normal (p: float3; n: float3) : float4*
- *plane_normalize (Plane: float4) : float4*

math_boost::**plane_dot(Plane: float4; Vec: float4) : float**()

dot product of *Plane* and 'Vec'

Arguments

- **Plane** : float4
- **Vec** : float4

math_boost::**plane_from_point_normal(p: float3; n: float3) : float4**()

construct plane from point *p* and normal *n*

Arguments

- **p** : float3
- **n** : float3

math_boost::**plane_normalize(Plane: float4) : float4**()

normalize `Plane`, length xyz will be 1.0 (or 0.0 for no plane)

Arguments

- **Plane** : float4

2.4.6 Color conversions

- *linear_to_SRGB (c: float3) : float3*
- *linear_to_SRGB (x: float) : float*
- *linear_to_SRGB (c: float4) : float4*

linear_to_SRGB

math_boost::**linear_to_SRGB(c: float3) : float3**()

convert value from linear space to sRGB curve space

Arguments

- **c** : float3

math_boost::**linear_to_SRGB(x: float) : float**()

math_boost::**linear_to_SRGB(c: float4) : float4**()

2.4.7 Color packing and unpacking

- *RGBA_TO_UCOLOR (xyzw: float4) : uint*
- *RGBA_TO_UCOLOR (x: float; y: float; z: float; w: float) : uint*
- *UCOLOR_TO_RGB (x: uint) : float3*
- *UCOLOR_TO_RGBA (x: uint) : float4*

RGBA_TO_UCOLOR

math_boost::**RGBA_TO_UCOLOR(xyzw: float4) : uint**()

conversion from RGBA to ucolor. xyzw components are in [0,1] range

Arguments

- **xyzw** : float4

math_boost::**RGBA_TO_UCOLOR(x: float; y: float; z: float; w: float) : uint**()

---

math_boost::**UCOLOR_TO_RGB(x: uint) : float3**()

conversion from ucolor to RGB. x components are in [0,255] range. result is float3(x,y,z)

Arguments

- **x** : uint

math_boost::**UCOLOR_TO_RGBA(x: uint) : float4**()

conversion from ucolor to RGBA. x components are in [0,255] range

Arguments

- **x** : uint

2.5 Random generator library

The RANDOM module implements pseudo-random number generation using a linear congruential generator with vectorized state (`int4`). It provides integer, float, and vector random values, as well as geometric sampling (unit vectors, points in spheres and disks).

All functions and symbols are in "random" module, use require to get access to it.

```
require daslib/random
```

Example:

```
require daslib/random

    [export]
    def main() {
        var seed = random_seed(12345)
        print("int: {random_int(seed)}\n")
        print("float: {random_float(seed)}\n")
        print("float: {random_float(seed)}\n")
    }
    // output:
    // int: 7584
    // float: 0.5848567
    // float: 0.78722495
```

2.5.1 Constants

`random::LCG_RAND_MAX = 32767`

maximum possible output of random number generator

`random::LCG_RAND_MAX_BIG = 1073741823`

maximum possible output of random_big_int

2.5.2 Seed and basic generators

- *random_big_int (var seed: int4&) : auto*
- *random_float (var seed: int4&) : auto*
- *random_float4 (var seed: int4&) : auto*
- *random_int (var seed: int4&) : auto*
- *random_int4 (var seed: int4&) : auto*
- *random_seed (seed: int) : auto*
- *random_seed2D (var seed: int4&; co: int2; cf: int = 0) : auto*
- *random_uint (var seed: int4&) : auto*

`random::`**`random_big_int(seed: int4&) : auto`**`()`

random integer 0..32768*32768-1 (LCG_RAND_MAX_BIG)

Arguments

- **seed** : int4&

`random::`**`random_float(seed: int4&) : auto`**`()`

random float 0..1

Arguments

- **seed** : int4&

`random::`**`random_float4(seed: int4&) : auto`**`()`

random float4, each component is 0..1

Arguments

- **seed** : int4&

`random::`**`random_int(seed: int4&) : auto`**`()`

random integer 0..32767 (LCG_RAND_MAX)

Arguments

- **seed** : int4&

`random::`**`random_int4(seed: int4&) : auto`**`()`

random int4, each component is 0..32767 (LCG_RAND_MAX)

Arguments

- **seed** : int4&

`random::`**`random_seed(seed: int) : auto`**`()`

constructs seed vector out of single integer seed

Arguments

- **seed** : int

`random::`**`random_seed2D(seed: int4&; co: int2; cf: int = 0) : auto`**`()`

constructs seed vector out of 2d screen coordinates and frame counter *cf*

Arguments

- **seed** : int4&
- **co** : int2
- **cf** : int

`random::`**`random_uint(seed: int4&) : auto`**`()`

random unsigned integer using 3-component LCG, covering full uint range

Arguments

- **seed** : int4&

2.5.3 Random iterators

- *each_random_uint (rnd_seed: int = 13) : iterator<uint>*

`random::`**`each_random_uint(rnd_seed: int = 13) : iterator<uint>`**`()`

infinite generator of random uints initialized with *rnd_seed*

Arguments

- **rnd_seed** : int

2.5.4 Specific distributions

- *random_in_unit_disk (var seed: int4&) : auto*
- *random_in_unit_sphere (var seed: int4&) : auto*
- *random_unit_vector (var seed: int4&) : auto*

`random::`**`random_in_unit_disk(seed: int4&) : auto`**`()`

Returns a random float3 point uniformly distributed inside the unit disk (length <= 1, z=0).

Arguments

- **seed** : int4&

`random::`**`random_in_unit_sphere(seed: int4&) : auto`**`()`

Returns a random float3 point uniformly distributed inside the unit sphere (length <= 1).

Arguments

- **seed** : int4&

`random::`**`random_unit_vector(seed: int4&) : auto`**`()`

random float3 unit vector (length=1.)

Arguments

- **seed** : int4&

CHAPTER

THREE

STRINGS

String manipulation, formatting, encoding, and temporary string utilities.

3.1 String manipulation library

The STRINGS module implements string formatting, conversion, searching, and modification routines. It provides functions for building strings (`build_string`), parsing (`to_int`, `to_float`), character classification (`is_alpha`, `is_number`), and low-level string manipulation.

All functions and symbols are in “strings” module, use require to get access to it.

```
require strings
```

3.1.1 Enumerations

strings::**ConversionResult**

Result of conversion from string to number.

Values

- **ok** = 0 - Successful conversion
- **invalid_argument** = 22 - Argument is not a valid number
- **out_of_range** = 34 - Argument is out of range for the target type

3.1.2 Handled structures

strings::**StringBuilderWriter**

Object representing a string builder. Its significantly faster to write data to the string builder and than convert it to a string, as oppose to using sequences of string concatenations.

3.1.3 Character set

- *is_char_in_set (Character: int; Charset: uint const[8]) : bool*
- *set_element (Character: int; Charset: uint const[8]) : int*
- *set_total (Charset: uint const[8]) : uint*

strings::**is_char_in_set(Character: int; Charset: uint const[8]) : bool**()

Returns true if the character given by its integer code is present in the 256-bit character set represented as a uint[8] array.

Arguments

- **Character** : int
- **Charset** : uint[8] implicit

strings::**set_element(Character: int; Charset: uint const[8]) : int**()

Returns the character code at the given element index within the 256-bit character set represented as a uint[8] array.

Arguments

- **Character** : int
- **Charset** : uint[8] implicit

strings::**set_total(Charset: uint const[8]) : uint**()

Returns the total number of characters present (bits set) in the 256-bit character set represented as a uint[8] array.

Arguments

- **Charset** : uint[8] implicit

3.1.4 Character groups

- *is_alnum (Character: int) : bool*
- *is_alpha (Character: int) : bool*
- *is_hex (Character: int) : bool*
- *is_new_line (Character: int) : bool*
- *is_number (Character: int) : bool*
- *is_tab_or_space (Character: int) : bool*
- *is_white_space (Character: int) : bool*

strings::**is_alnum(Character: int) : bool**()

Returns true if the integer character code represents an alphanumeric ASCII character [A-Za-z0-9].

Arguments

- **Character** : int

strings::**is_alpha(Character: int) : bool**()

Returns true if the integer character code represents an alphabetic ASCII character [A-Za-z].

Arguments

- **Character** : int

`strings::`**`is_hex(Character: int) : bool`**`()`

Returns true if the integer character code represents a hexadecimal digit [0-9A-Fa-f].

Arguments

- **Character** : int

`strings::`**`is_new_line(Character: int) : bool`**`()`

Returns true if the integer character code is a newline character (\n or \r).

Arguments

- **Character** : int

`strings::`**`is_number(Character: int) : bool`**`()`

Returns true if the integer character code represents a decimal digit [0-9].

Arguments

- **Character** : int

`strings::`**`is_tab_or_space(Character: int) : bool`**`()`

Returns true if the integer character code is a tab or space character.

Arguments

- **Character** : int

`strings::`**`is_white_space(Character: int) : bool`**`()`

Returns true if the integer character code is a whitespace character (space, tab, newline, carriage return, etc.).

Arguments

- **Character** : int

3.1.5 Character by index

- *character_at (str: string; idx: int) : int*
- *character_uat (str: string; idx: int) : int*

`strings::`**`character_at(str: string; idx: int) : int`**`()`

Returns the integer character code of string *str* at the given index *idx*, with bounds checking.

Arguments

- **str** : string implicit
- **idx** : int

`strings::`**`character_uat(str: string; idx: int) : int`**`()`

Warning: This is unsafe operation.

Returns the integer character code of string *str* at the given index *idx* without performing bounds checking (unsafe).

Arguments

- **str** : string implicit
- **idx** : int

3.1.6 String properties

- *ends_with (str: das_string; cmp: string) : bool*
- *ends_with (str: string; cmp: string) : bool*
- *length (str: string) : int*
- *length (str: das_string) : int*
- *starts_with (str: string; cmp: string; cmpLen: uint) : bool*
- *starts_with (str: string; cmp: string) : bool*
- *starts_with (str: string; offset: int; cmp: string) : bool*
- *starts_with (str: string; offset: int; cmp: string; cmpLen: uint) : bool*

ends_with

`strings::`**`ends_with(str: das_string; cmp: string) : bool`**`()`

Returns true if the string *str* ends with the substring *cmp*, false otherwise.

Arguments

- **str** : *das_string* implicit
- **cmp** : string implicit

`strings::`**`ends_with(str: string; cmp: string) : bool`**`()`

---

length

`strings::`**`length(str: string) : int`**`()`

Returns the length of the string or das_string in characters as an int.

Arguments

- **str** : string implicit

`strings::`**`length(str: das_string) : int`**`()`

---

starts_with

strings::**starts_with(str: string; cmp: string; cmpLen: uint) : bool**()

Returns true if the beginning of string *str* matches the string *cmp*, with optional *offset* and *cmpLen* parameters to control the comparison start position and length.

Arguments

- **str** : string implicit
- **cmp** : string implicit
- **cmpLen** : uint

strings::**starts_with(str: string; cmp: string) : bool**()

strings::**starts_with(str: string; offset: int; cmp: string) : bool**()

strings::**starts_with(str: string; offset: int; cmp: string; cmpLen: uint) : bool**()

3.1.7 String builder

- *build_hash (block: block<(StringBuilderWriter):void>) : uint64*
- *build_string (block: block<(StringBuilderWriter):void>) : string*
- *format (format: string; value: int64) : string*
- *format (format: string; value: double) : string*
- *format (writer: StringBuilderWriter; format: string; value: int) : StringBuilderWriter&*
- *format (writer: StringBuilderWriter; format: string; value: uint) : StringBuilderWriter&*
- *format (writer: StringBuilderWriter; format: string; value: int64) : StringBuilderWriter&*
- *format (format: string; value: uint64) : string*
- *format (writer: StringBuilderWriter; format: string; value: float) : StringBuilderWriter&*
- *format (writer: StringBuilderWriter; format: string; value: uint64) : StringBuilderWriter&*
- *format (writer: StringBuilderWriter; format: string; value: double) : StringBuilderWriter&*
- *format (format: string; value: int) : string*
- *format (format: string; value: uint) : string*
- *format (format: string; value: float) : string*
- *write (writer: StringBuilderWriter; anything: any) : StringBuilderWriter&*
- *write_char (writer: StringBuilderWriter; ch: int) : StringBuilderWriter&*
- *write_chars (writer: StringBuilderWriter; ch: int; count: int) : StringBuilderWriter&*
- *write_escape_string (writer: StringBuilderWriter; str: string) : StringBuilderWriter&*

strings::**build_hash(block: block<(StringBuilderWriter):void>) : uint64**()

Computes a uint64 hash by streaming writes through a StringBuilderWriter passed to *block*, without allocating the full concatenated string.

Arguments

- **block** : block<(*StringBuilderWriter*):void> implicit

`strings::build_string(block: block<(StringBuilderWriter):void>) : string()`

Creates a StringBuilderWriter, passes it to *block* for writing, and returns the accumulated output as a string.

Arguments

- **block** : block<(*StringBuilderWriter*):void> implicit

format

`strings::format(format: string; value: int64) : string()`

Warning: This function is deprecated.

Formats a numeric value of type T using a C printf-style format string, either appending to a StringBuilderWriter and returning a reference to it, or returning the formatted result as a new string.

Arguments

- **format** : string implicit
- **value** : int64

`strings::format(format: string; value: double) : string()`

`strings::format(writer: StringBuilderWriter; format: string; value: int) : StringBuilderWriter&()`

`strings::format(writer: StringBuilderWriter; format: string; value: uint) : StringBuilderWriter&()`

`strings::format(writer: StringBuilderWriter; format: string; value: int64) : StringBuilderWriter&()`

`strings::format(format: string; value: uint64) : string()`

`strings::format(writer: StringBuilderWriter; format: string; value: float) : StringBuilderWriter&()`

`strings::format(writer: StringBuilderWriter; format: string; value: uint64) : StringBuilderWriter&()`

`strings::format(writer: StringBuilderWriter; format: string; value: double) : StringBuilderWriter&()`

`strings::format(format: string; value: int) : string()`

`strings::format(format: string; value: uint) : string()`

`strings::format(format: string; value: float) : string()`

`strings::write(writer: StringBuilderWriter; anything: any) : StringBuilderWriter&()`

Writes the textual representation of any value into the StringBuilderWriter and returns a reference to the writer for chaining.

Arguments

- **writer** : *StringBuilderWriter*
- **anything** : any

strings::**write_char(writer: StringBuilderWriter; ch: int) : StringBuilderWriter&**()

Writes a single character specified by its integer code *ch* into the StringBuilderWriter and returns a reference to the writer.

Arguments

- **writer** : *StringBuilderWriter* implicit
- **ch** : int

strings::**write_chars(writer: StringBuilderWriter; ch: int; count: int) : StringBuilderWriter&**()

Writes the character specified by integer code *ch* repeated *count* times into the StringBuilderWriter and returns a reference to the writer.

Arguments

- **writer** : *StringBuilderWriter* implicit
- **ch** : int
- **count** : int

strings::**write_escape_string(writer: StringBuilderWriter; str: string) : StringBuilderWriter&**()

Writes the escaped form of string *str* (with special characters converted to escape sequences) into the StringBuilderWriter and returns a reference to the writer.

Arguments

- **writer** : *StringBuilderWriter* implicit
- **str** : string implicit

3.1.8 das::string manipulation

- *append (str: das_string; ch: int)*
- *resize (str: das_string; new_length: int)*

strings::**append**(*str: das_string; ch: int*)

Appends a single character specified by its integer code *ch* to the mutable das_string *str*.

Arguments

- **str** : *das_string* implicit
- **ch** : int

strings::**resize**(*str: das_string; new_length: int*)

Resizes the mutable das_string *str* in place to *new_length* characters.

Arguments

- **str** : *das_string* implicit
- **new_length** : int

3.1.9 String modifications

- *chop (str: string; start: int; length: int) : string*
- *escape (str: string) : string*
- *ltrim (str: string) : string*
- *repeat (str: string; count: int) : string*
- *replace (str: string; toSearch: string; replace: string) : string*
- *reverse (str: string) : string*
- *rtrim (str: string; chars: string) : string*
- *rtrim (str: string) : string*
- *safe_unescape (str: string) : string*
- *slice (str: string; start: int) : string*
- *slice (str: string; start: int; end: int) : string*
- *strip (str: string) : string*
- *strip_left (str: string) : string*
- *strip_right (str: string) : string*
- *to_lower (str: string) : string*
- *to_lower_in_place (str: string) : string*
- *to_upper (str: string) : string*
- *to_upper_in_place (str: string) : string*
- *trim (str: string) : string*
- *unescape (str: string) : string*

`strings::`**`chop(str: string; start: int; length: int) : string`**`()`

Returns a substring of *str* beginning at index *start* with the specified *length*.

Arguments

- **str** : string implicit
- **start** : int
- **length** : int

`strings::`**`escape(str: string) : string`**`()`

Returns a new string with special characters replaced by their printable escape sequences (e.g. newline becomes \n).

Arguments

- **str** : string implicit

`strings::`**`ltrim(str: string) : string`**`()`

Returns a new string with leading whitespace characters removed from *str*.

Arguments

- **str** : string implicit

strings::**repeat(str: string; count: int) : string**()

Returns a new string formed by concatenating *str* repeated *count* times.

Arguments

- **str** : string implicit
- **count** : int

strings::**replace(str: string; toSearch: string; replace: string) : string**()

Returns a new string with all occurrences of substring *toSearch* in *str* replaced by the substring *replace*.

Arguments

- **str** : string implicit
- **toSearch** : string implicit
- **replace** : string implicit

strings::**reverse(str: string) : string**()

Returns a new string with the characters of *str* in reverse order.

Arguments

- **str** : string implicit

rtrim

strings::**rtrim(str: string; chars: string) : string**()

Returns a new string with trailing whitespace removed from *str*, or with trailing characters from the specified *chars* set removed.

Arguments

- **str** : string implicit
- **chars** : string implicit

strings::**rtrim(str: string) : string**()

---

strings::**safe_unescape(str: string) : string**()

Unescapes a string by converting printable escape sequences back to their original characters (e.g. \n becomes a new-line), skipping invalid sequences instead of failing.

Arguments

- **str** : string implicit

slice

`strings::slice(str: string; start: int) : string()`

Returns a substring of *str* from index *start* to optional *end* (exclusive), where negative indices count from the end of the string.

Arguments

- **str** : string implicit
- **start** : int

`strings::slice(str: string; start: int; end: int) : string()`

`strings::strip(str: string) : string()`

Returns a new string with all leading and trailing whitespace characters removed from *str*.

Arguments

- **str** : string implicit

`strings::strip_left(str: string) : string()`

Returns a new string with all leading whitespace characters removed from *str*.

Arguments

- **str** : string implicit

`strings::strip_right(str: string) : string()`

Returns a new string with all trailing whitespace characters removed from *str*.

Arguments

- **str** : string implicit

`strings::to_lower(str: string) : string()`

Returns a new string with all characters of *str* converted to lower case.

Arguments

- **str** : string implicit

`strings::to_lower_in_place(str: string) : string()`

Warning: This is unsafe operation.

Converts all characters of *str* to lower case in place and returns the modified string.

Arguments

- **str** : string implicit

`strings::`**`to_upper(str: string) : string`**`()`

Returns a new string with all characters of *str* converted to upper case.

Arguments

- **str** : string implicit

`strings::`**`to_upper_in_place(str: string) : string`**`()`

Warning: This is unsafe operation.

Converts all characters of *str* to upper case in place and returns the modified string.

Arguments

- **str** : string implicit

`strings::`**`trim(str: string) : string`**`()`

Returns a new string with both leading and trailing whitespace characters removed from *str*.

Arguments

- **str** : string implicit

`strings::`**`unescape(str: string) : string`**`()`

Returns a new string with printable escape sequences converted back to their original characters (e.g. \n becomes a newline).

Arguments

- **str** : string implicit

3.1.10 Search substrings

- *find (str: string; substr: string) : int*
- *find (str: string; substr: string; start: int) : int*
- *find (str: string; substr: int; start: int) : int*
- *find (str: string; substr: int) : int*

find

`strings::`**`find(str: string; substr: string) : int`**`()`

Returns the first index at which *substr* (string or character code) occurs in *str*, optionally searching from *start*, or -1 if not found.

Arguments

- **str** : string implicit
- **substr** : string implicit

`strings::`**`find(str: string; substr: string; start: int) : int`**`()`

`strings::`**`find(str: string; substr: int; start: int) : int`**`()`

`strings::`**`find(str: string; substr: int) : int`**`()`

3.1.11 String comparison

- *compare_ignore_case (a: string; b: string) : int*

`strings::`**`compare_ignore_case(a: string; b: string) : int`**`()`

Performs case-insensitive string comparison. Returns 0 if strings are equal, a negative value if *a* is less than *b*, or a positive value if *a* is greater than *b*.

Arguments

- **a** : string implicit
- **b** : string implicit

3.1.12 String conversion routines

- *double (str: string; result: ConversionResult&; offset: int&) : double*
- *double (str: string) : double*
- *float (str: string) : float*
- *float (str: string; result: ConversionResult&; offset: int&) : float*
- *fmt (writer: StringBuilderWriter; format: string; value: int64) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: uint) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: uint64) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: uint16) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: uint8) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: int8) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: int16) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: int) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: double) : StringBuilderWriter&*
- *fmt (writer: StringBuilderWriter; format: string; value: float) : StringBuilderWriter&*
- *int (str: string) : int*
- *int (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int*
- *int16 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int16*
- *int16 (str: string) : int16*
- *int64 (str: string) : int64*
- *int64 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int64*
- *int8 (str: string) : int8*
- *int8 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int8*

- *string (bytes: array<uint8>) : string*
- *to_char (char: int) : string*
- *to_cpp_float (value: float) : string*
- *to_double (value: string) : double*
- *to_float (value: string) : float*
- *to_int (value: string; hex: bool = false) : int*
- *to_int16 (value: string; hex: bool = false) : int16*
- *to_int64 (value: string; hex: bool = false) : int64*
- *to_int8 (value: string; hex: bool = false) : int8*
- *to_uint (value: string; hex: bool = false) : uint*
- *to_uint16 (value: string; hex: bool = false) : uint16*
- *to_uint64 (value: string; hex: bool = false) : uint64*
- *to_uint8 (value: string; hex: bool = false) : uint8*
- *uint (str: string) : uint*
- *uint (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint*
- *uint16 (str: string) : uint16*
- *uint16 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint16*
- *uint64 (str: string) : uint64*
- *uint64 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint64*
- *uint8 (str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint8*
- *uint8 (str: string) : uint8*

double

`strings::double(str: string; result: ConversionResult&; offset: int&) : double()`

Converts a string to a double value, panicking on failure; an overload accepts *result* and *offset* output parameters to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit
- **result** : *ConversionResult&* implicit
- **offset** : int& implicit

`strings::double(str: string) : double()`

float

`strings::float(str: string) : float()`

Converts a string to a float value, panicking on failure; an overload accepts *result* and *offset* output parameters to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

`strings::float(str: string; result: ConversionResult&; offset: int&) : float()`

---

fmt

`strings::fmt(writer: StringBuilderWriter; format: string; value: int64) : StringBuilderWriter&()`

Formats a numeric value of type T into the StringBuilderWriter using a libfmt/C++20 std::format format string and returns a reference to the writer.

Arguments

- **writer** : *StringBuilderWriter* implicit
- **format** : string implicit
- **value** : int64

`strings::fmt(writer: StringBuilderWriter; format: string; value: uint) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: uint64) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: uint16) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: uint8) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: int8) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: int16) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: int) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: double) : StringBuilderWriter&()`

`strings::fmt(writer: StringBuilderWriter; format: string; value: float) : StringBuilderWriter&()`

---

int

`strings::int(str: string) : int()`

Converts a string to an int, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**int(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int()**

---

int16

strings::**int16(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int16()**

Converts a string to an int16, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit
- **result** : *ConversionResult*& implicit
- **offset** : int& implicit
- **hex** : bool

strings::**int16(str: string) : int16()**

---

int64

strings::**int64(str: string) : int64()**

Converts a string to an int64, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**int64(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int64()**

---

int8

strings::**int8(str: string) : int8()**

Converts a string to an int8, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**int8(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : int8()**

---

strings::**string(bytes: array<uint8>) : string**()

Constructs and returns a new string from the contents of a uint8 byte array.

Arguments

- **bytes** : array<uint8> implicit

strings::**to_char(char: int) : string**()

Converts an integer character code to a single-character string.

Arguments

- **char** : int

strings::**to_cpp_float(value: float) : string**()

Converts a float value to its string representation using C++ fmt::format_to, correctly handling special constants like FLT_MIN and FLT_MAX.

Arguments

- **value** : float

strings::**to_double(value: string) : double**()

Converts a string to a double value, returning 0.0lf if the conversion fails.

Arguments

- **value** : string implicit

strings::**to_float(value: string) : float**()

Converts a string to a float value, returning 0.0 if the conversion fails.

Arguments

- **value** : string implicit

strings::**to_int(value: string; hex: bool = false) : int**()

Converts a string to an int value with optional hexadecimal parsing when *hex* is true, returning 0 if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

strings::**to_int16(value: string; hex: bool = false) : int16**()

Converts a string to an int16 value with optional hexadecimal parsing when *hex* is true, returning 0 if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

strings::**to_int64(value: string; hex: bool = false) : int64**()

Converts a string to an int64 value with optional hexadecimal parsing when *hex* is true, returning 0l if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

`strings::to_int8(value: string; hex: bool = false) : int8()`

Converts a string to an int8 value with optional hexadecimal parsing when *hex* is true, returning 0 if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

`strings::to_uint(value: string; hex: bool = false) : uint()`

Converts a string to a uint value with optional hexadecimal parsing when *hex* is true, returning 0u if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

`strings::to_uint16(value: string; hex: bool = false) : uint16()`

Converts a string to a uint16 value. Returns 0 if conversion fails. When *hex* is true, parses the string as hexadecimal.

Arguments

- **value** : string implicit
- **hex** : bool

`strings::to_uint64(value: string; hex: bool = false) : uint64()`

Converts a string to a uint64 value with optional hexadecimal parsing when *hex* is true, returning 0ul if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

`strings::to_uint8(value: string; hex: bool = false) : uint8()`

Converts a string to a uint8 value with optional hexadecimal parsing when *hex* is true, returning 0u if the conversion fails.

Arguments

- **value** : string implicit
- **hex** : bool

uint

strings::**uint(str: string) : uint**()

Converts a string to a uint, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**uint(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint**()

---

uint16

strings::**uint16(str: string) : uint16**()

Converts a string to a uint16, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**uint16(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint16**()

---

uint64

strings::**uint64(str: string) : uint64**()

Converts a string to a uint64, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit

strings::**uint64(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint64**()

---

uint8

strings::**uint8(str: string; result: ConversionResult&; offset: int&; hex: bool = false) : uint8**()

Converts a string to a uint8, panicking on failure; an overload accepts *result*, *offset*, and optional *hex* flag to report the ConversionResult status and parsed position instead of panicking.

Arguments

- **str** : string implicit
- **result** : *ConversionResult*& implicit
- **offset** : int& implicit
- **hex** : bool

strings::**uint8(str: string) : uint8**()

3.1.13 String as array

- *modify_data (str: string; block: block<(array<uint8>#):void>) : string*
- *peek_data (str: string; block: block<(array<uint8>#):void>)*

strings::**modify_data(str: string; block: block<(array<uint8>#):void>) : string**()

Maps the raw bytes of string *str* into a temporary uint8 array, passes it to *block* for in-place reading and writing, and returns the modified string.

Arguments

- **str** : string implicit
- **block** : block<(array<uint8>#):void> implicit

strings::**peek_data**(*str: string; block: block<(array<uint8>#):void>*)

Maps the raw bytes of string *str* into a temporary read-only uint8 array and passes it to *block* for inspection.

Arguments

- **str** : string implicit
- **block** : block<(array<uint8>#):void> implicit

3.1.14 Low level memory allocation

- *delete_string (str: string&) : bool*
- *reserve_string_buffer (str: string; length: int) : string*

strings::**delete_string(str: string&) : bool**()

Warning: This is unsafe operation.

Frees the string *str* from the heap and clears the reference, returning true on success; unsafe because existing aliases become dangling pointers.

Arguments

- **str** : string& implicit

strings::**reserve_string_buffer(str: string; length: int) : string**()

Allocates a copy of the string data on the heap with at least *length* bytes reserved and returns the new string.

Arguments

- **str** : string implicit
- **length** : int

3.2 Boost package for string manipulation library

The STRINGS_BOOST module extends string handling with splitting, joining, padding, character replacement, and edit distance computation.

All functions and symbols are in "strings_boost" module, use require to get access to it.

```
require daslib/strings_boost
```

Example:

```
require daslib/strings_boost

    [export]
    def main() {
        let parts = split("one,two,three", ",")
        print("split: {parts}\n")
        print("join: {join(parts, " | ")}\n")
        print("[{wide("hello", 10)}]\n")
        print("distance: {levenshtein_distance("kitten", "sitting")}\n")
    }
    // output:
    // split: [[ one; two; three]]
    // join: one | two | three
    // [hello     ]
    // distance: 3
```

3.2.1 Split and join

- *join (it: auto; separator: string) : auto*
- *join (iterable: array<auto(TT)>; separator: string; blk: block<(var writer:StringBuilderWriter;elem:TT):void>) : string*
- *join (var it: iterator<auto(TT)>; separator: string) : auto*
- *join (var iterable: iterator<auto(TT)>; separator: string; blk: block<(var writer:StringBuilderWriter;elem:TT):void>) : string*
- *join (iterable: auto(TT)[]; separator: string; blk: block<(var writer:StringBuilderWriter;elem:TT):void>) : string*
- *split (text: string; delim: string) : array<string>*
- *split (text: string; delim: string; blk: block<(arg:array<string>#):auto>) : auto*
- *split_by_chars (text: string; delim: string) : array<string>*
- *split_by_chars (text: string; delim: string; blk: block<(arg:array<string>#):auto>) : auto*

join

`strings_boost::join(it: auto; separator: string) : auto()`

Joins the elements of an iterable into a single string using the specified separator.

Arguments

- **it** : auto
- **separator** : string implicit

`strings_boost::join(iterable: array<auto(TT)>; separator: string; blk: block<(var writer:StringBuilderW`

`strings_boost::join(it: iterator<auto(TT)>; separator: string) : auto()`

`strings_boost::join(iterable: iterator<auto(TT)>; separator: string; blk: block<(var writer:StringBuild`

`strings_boost::join(iterable: auto(TT)[]; separator: string; blk: block<(var writer:StringBuilderWriter`

---

split

`strings_boost::split(text: string; delim: string) : array<string>()`

Splits a string by the specified delimiter string and returns an array of substrings.

Arguments

- **text** : string implicit
- **delim** : string implicit

`strings_boost::split(text: string; delim: string; blk: block<(arg:array<string>#):auto>) : auto()`

---

split_by_chars

`strings_boost::split_by_chars(text: string; delim: string) : array<string>()`

Splits a string by the specified delimiter characters and returns an array of substrings.

Arguments

- **text** : string implicit
- **delim** : string implicit

`strings_boost::split_by_chars(text: string; delim: string; blk: block<(arg:array<string>#):auto>) : aut`

3.2.2 Formatting

- *capitalize (str: string) : string*
- *pad_left (str: string; width: int; ch: int = 32) : string*
- *pad_right (str: string; width: int; ch: int = 32) : string*
- *wide (text: string; width: int) : string*

`strings_boost::`**`capitalize(str: string) : string`**`()`

Returns a copy of the string with the first character converted to uppercase. The rest of the string is unchanged.

Arguments

- **str** : string

`strings_boost::`**`pad_left(str: string; width: int; ch: int = 32) : string`**`()`

Pads the string with the character *ch* on the left to reach the specified minimum *width*. If the string is already at least *width* characters, it is returned unchanged.

Arguments

- **str** : string
- **width** : int
- **ch** : int

`strings_boost::`**`pad_right(str: string; width: int; ch: int = 32) : string`**`()`

Pads the string with the character *ch* on the right to reach the specified minimum *width*. If the string is already at least *width* characters, it is returned unchanged.

Arguments

- **str** : string
- **width** : int
- **ch** : int

`strings_boost::`**`wide(text: string; width: int) : string`**`()`

Pads the string with trailing spaces to reach the specified minimum width.

Arguments

- **text** : string implicit
- **width** : int

3.2.3 Queries and comparisons

- *contains (str: string; sub: string) : bool*
- *count (str: string; sub: string) : int*
- *eq (b: das_string; a: string) : auto*
- *eq (a: string; b: das_string) : auto*
- *is_character_at (foo: array<uint8>; idx: int; ch: int) : auto*
- *is_null_or_whitespace (str: string) : bool*

`strings_boost::contains(str: string; sub: string) : bool()`

Returns true if *sub* is found anywhere within *str*.

Arguments

- **str** : string
- **sub** : string

`strings_boost::count(str: string; sub: string) : int()`

Counts non-overlapping occurrences of *sub* in *str*. Returns 0 if *sub* is empty or not found.

Arguments

- **str** : string
- **sub** : string

eq

`strings_boost::eq(b: das_string; a: string) : auto()`

Compares a `string` with a `das_string` for equality, returning `true` if they match.

Arguments

- **b** : *das_string*
- **a** : string implicit

`strings_boost::eq(a: string; b: das_string) : auto()`

`strings_boost::is_character_at(foo: array<uint8>; idx: int; ch: int) : auto()`

Returns `true` if the byte at the specified index in the array equals the given character code.

Arguments

- **foo** : array<uint8> implicit
- **idx** : int
- **ch** : int

`strings_boost::is_null_or_whitespace(str: string) : bool()`

Returns true if the string is null, empty, or contains only whitespace characters (space, tab, CR, LF).

Arguments

- **str** : string

3.2.4 Search

- *last_index_of (str: string; sub: string; start: int) : int*
- *last_index_of (str: string; sub: string) : int*

last_index_of

`strings_boost::`**`last_index_of(str: string; sub: string; start: int) : int`**`()`

Returns the index of the last occurrence of *sub* in *str* searching only up to position *start* (exclusive), or -1 if not found.

Arguments

- **str** : string
- **sub** : string
- **start** : int

`strings_boost::`**`last_index_of(str: string; sub: string) : int`**`()`

3.2.5 Replace

- *replace_multiple (source: string; replaces: array<tuple<text:string;replacement:string>>) : string*

`strings_boost::`**`replace_multiple(source: string; replaces: array<tuple<text:string;replacement:string>>)`**

Applies multiple find-and-replace substitutions to a string in a single pass.

Arguments

- **source** : string
- **replaces** : array<tuple<text:string;replacement:string>>

3.2.6 Prefix and suffix

- *trim_prefix (str: string; prefix: string) : string*
- *trim_suffix (str: string; suffix: string) : string*

`strings_boost::`**`trim_prefix(str: string; prefix: string) : string`**`()`

Removes *prefix* from the beginning of *str* if present. Returns the string unchanged if it does not start with *prefix*.

Arguments

- **str** : string
- **prefix** : string

`strings_boost::`**`trim_suffix(str: string; suffix: string) : string`**`()`

Removes *suffix* from the end of *str* if present. Returns the string unchanged if it does not end with *suffix*.

Arguments

- **str** : string
- **suffix** : string

3.2.7 Levenshtein distance

- *levenshtein_distance (s: string; t: string) : int*
- *levenshtein_distance_fast (s: string; t: string) : int*

`strings_boost::`**`levenshtein_distance(s: string; t: string) : int()`**

Computes the Levenshtein edit distance between two strings.

Arguments

- **s** : string implicit
- **t** : string implicit

`strings_boost::`**`levenshtein_distance_fast(s: string; t: string) : int()`**

Computes the Levenshtein edit distance between two strings using an optimized algorithm.

Arguments

- **s** : string implicit
- **t** : string implicit

3.3 Temporary string utilities

The TEMP_STRINGS module provides temporary string construction that avoids heap allocations. Temporary strings are allocated on the stack or in scratch memory and are valid only within the current scope, offering fast string building for formatting and output.

All functions and symbols are in "temp_strings" module, use require to get access to it.

```
require daslib/temp_strings
```

3.3.1 Function annotations

`temp_strings::`**`TempStringMacro`**

Function annotation that enables temporary string optimization.

3.3.2 Temporary string builders

- *build_temp_string (bldr: block<(var writer:StringBuilderWriter):void>; cb: block<(res:string#):void>)*

`temp_strings::`**`build_temp_string`**`(`*bldr: block<(var writer:StringBuilderWriter):void>; cb: block<(res:string#):void>*`)`

Same as build_string, but delete the string after the callback is called. Intern strings are not deleted.

Arguments

- **bldr** : block<(writer: *StringBuilderWriter*):void>
- **cb** : block<(res:string#):void>

3.3.3 Temporary string conversion

- *temp_string (arr: array<uint8>; cb: block<(res:string#):void>)*
- *temp_string (str: string; cb: block<(res:string#):void>)*

temp_string

`temp_strings::`**`temp_string`**`(`*arr: array<uint8>; cb: block<(res:string#):void>*`)`

Construct string from array of bytes and pass it to the callback. Delete the string after the callback is called. Intern strings are not deleted.

Arguments

- **arr** : array<uint8>
- **cb** : block<(res:string#):void>

`temp_strings::`**`temp_string`**`(`*str: string; cb: block<(res:string#):void>*`)`

3.4 UTF-8 utilities

The UTF8_UTILS module provides Unicode UTF-8 string utilities including character iteration, codepoint extraction, byte length calculation, and validation of UTF-8 encoded text.

All functions and symbols are in “utf8_utils” module, use require to get access to it.

```
require daslib/utf8_utils
```

3.4.1 Constants

`utf8_utils::`**`s_utf8d = fixed_array<uint>`**

Byte-class and state-transition table for the UTF-8 DFA decoder.

`utf8_utils::`**`UTF8_ACCEPT = 0x0`**

DFA accept state indicating a valid UTF-8 sequence.

3.4.2 Encoding and decoding

- *decode_unicode_escape (str: string) : string*
- *utf16_to_utf32 (high: uint; low: uint) : uint*
- *utf8_decode (source_utf8_string: string) : array<uint>*
- *utf8_decode (var dest_utf32_string: array<uint>; source_utf8_string: string)*
- *utf8_decode (source_utf8_string: array<uint8>) : array<uint>*
- *utf8_decode (var dest_utf32_string: array<uint>; source_utf8_string: array<uint8>)*
- *utf8_encode (var dest_array: array<uint8>; source_utf32_string: array<uint>)*
- *utf8_encode (var dest_array: array<uint8>; ch: uint)*

- *utf8_encode (ch: uint) : array<uint8>*
- *utf8_encode (source_utf32_string: array<uint>) : array<uint8>*

`utf8_utils::`**`decode_unicode_escape(str: string) : string`**`()`

Decodes Unicode escape sequences (backslash followed by hex digits) in a string to UTF-8.

Arguments

- **str** : string

`utf8_utils::`**`utf16_to_utf32(high: uint; low: uint) : uint`**`()`

Converts a UTF-16 surrogate pair to a single UTF-32 codepoint.

Arguments

- **high** : uint
- **low** : uint

utf8_decode

`utf8_utils::`**`utf8_decode(source_utf8_string: string) : array<uint>`**`()`

Converts UTF-8 string to UTF-32 and returns it as an array of codepoints (UTF-32 string)

Arguments

- **source_utf8_string** : string

`utf8_utils::`**`utf8_decode`**`(`*dest_utf32_string: array<uint>; source_utf8_string: string*`)`

`utf8_utils::`**`utf8_decode(source_utf8_string: array<uint8>) : array<uint>`**`()`

`utf8_utils::`**`utf8_decode`**`(`*dest_utf32_string: array<uint>; source_utf8_string: array<uint8>*`)`

---

utf8_encode

`utf8_utils::`**`utf8_encode`**`(`*dest_array: array<uint8>; source_utf32_string: array<uint>*`)`

Converts UTF-32 string to UTF-8 and appends it to the UTF-8 byte array

Arguments

- **dest_array** : array<uint8>
- **source_utf32_string** : array<uint> implicit

`utf8_utils::`**`utf8_encode`**`(`*dest_array: array<uint8>; ch: uint*`)`

`utf8_utils::`**`utf8_encode(ch: uint) : array<uint8>`**`()`

`utf8_utils::`**`utf8_encode(source_utf32_string: array<uint>) : array<uint8>`**`()`

3.4.3 Length and measurement

- *utf8_length (utf8_string: string) : int*
- *utf8_length (utf8_string: array<uint8>) : int*

utf8_length

`utf8_utils::`**`utf8_length(utf8_string: string) : int`**`()`

Returns the number of characters in the UTF-8 string

Arguments

- **utf8_string** : string

`utf8_utils::`**`utf8_length(utf8_string: array<uint8>) : int`**`()`

3.4.4 Validation

- *contains_utf8_bom (utf8_string: array<uint8>) : bool*
- *contains_utf8_bom (utf8_string: string) : bool*
- *is_first_byte_of_utf8_char (ch: uint8) : bool*
- *is_utf8_string_valid (utf8_string: array<uint8>) : bool*
- *is_utf8_string_valid (utf8_string: string) : bool*

contains_utf8_bom

`utf8_utils::`**`contains_utf8_bom(utf8_string: array<uint8>) : bool`**`()`

Returns true if the byte array starts with a UTF-8 BOM (byte order mark).

Arguments

- **utf8_string** : array<uint8> implicit

`utf8_utils::`**`contains_utf8_bom(utf8_string: string) : bool`**`()`

`utf8_utils::`**`is_first_byte_of_utf8_char(ch: uint8) : bool`**`()`

Returns true if the given byte is the first byte of a UTF-8 character.

Arguments

- **ch** : uint8

is_utf8_string_valid

utf8_utils::**is_utf8_string_valid(utf8_string: array<uint8>) : bool**()

Returns true if the byte array contains a valid UTF-8 encoded string.

Arguments

- **utf8_string** : array<uint8> implicit

utf8_utils::**is_utf8_string_valid(utf8_string: string) : bool**()

3.5 Base64 encoding and decoding

The BASE64 module implements Base64 encoding and decoding. It provides `base64_encode` and `base64_decode` for converting between binary data (strings or `array<uint8>`) and Base64 text representation.

All functions and symbols are in "base64" module, use require to get access to it.

```
require daslib/base64
```

Example:

```
require daslib/base64

    [export]
    def main() {
        let encoded = base64_encode("Hello, daslang!")
        print("encoded: {encoded}\n")
        let decoded = base64_decode(encoded)
        print("decoded: {decoded.text}\n")
    }
    // output:
    // encoded: SGVsbG8sIGRhU2NyaXB0IQ==
    // decoded: Hello, daslang!
```

3.5.1 Encoding

- *BASE64_ENCODE_OUT_SIZE (s: int) : int*
- *base64_encode (_inp: string) : string*
- *base64_encode (inp: array<uint8>|array<uint8>#) : auto*

base64::**BASE64_ENCODE_OUT_SIZE(s: int) : int**()

Returns the encoded output size for binary data of length *s*.

Arguments

- **s** : int

base64_encode

base64::**base64_encode(_inp: string) : string**()

Encodes a string to its Base64 text representation.

Arguments

- **_inp** : string

base64::**base64_encode(inp: array<uint8>|array<uint8>#) : auto**()

3.5.2 Decoding

- *BASE64_DECODE_OUT_SIZE (s: int) : int*
- *base64_decode (_in: string) : tuple<text:string;size:int>*
- *base64_decode (_in: string; var out: array<uint8>) : int*

base64::**BASE64_DECODE_OUT_SIZE(s: int) : int**()

Returns the maximum decoded output size for a Base64 string of length *s*.

Arguments

- **s** : int

base64_decode

base64::**base64_decode(_in: string) : tuple<text:string;size:int>**()

Decodes a Base64-encoded string. Returns a tuple of the decoded text and its byte length.

Arguments

- **_in** : string

base64::**base64_decode(_in: string; out: array<uint8>) : int**()

3.6 Long string embedding macro

The STRINGIFY module provides the `%stringify~` reader macro for embedding multi-line string literals verbatim. Text between `%stringify~` and `%%` is captured as-is without requiring escape sequences for quotes, braces, or other special characters.

All functions and symbols are in “stringify” module, use require to get access to it.

```
require daslib/stringify
```

3.6.1 Reader macros

`stringify::`**`stringify`**

This macro implements embedding of the long string into the source code.

```
var st = %stringify~
This is a long string
with multiple lines
and special charactes like { this } and "that"
%%
```

CHAPTER

FOUR

I/O AND SERIALIZATION

File I/O, networking, URI parsing, and binary serialization.

4.1 File input output library

The FIO module implements file input/output and filesystem operations. It provides functions for reading and writing files (`fopen`, `fread`, `fwrite`), directory management (`mkdir`, `dir`), path manipulation (`join_path`, `basename`, `dirname`), and file metadata queries (`stat`, `file_time`).

All functions and symbols are in "fio" module, use require to get access to it.

```
require fio
```

Example:

```
require fio

    [export]
    def main() {
        let fname = "_test_fio_tmp.txt"
        fopen(fname, "wb") $(f) {
            fwrite(f, "hello, daslang!")
        }
        fopen(fname, "rb") $(f) {
            let content = fread(f)
            print("{content}\n")
        }
        remove(fname)
    }
    // output:
    // hello, daslang!
```

4.1.1 Type aliases

`fio::file = FILE const?`

Type alias for FILE const? used as the standard file handle parameter type across fio functions.

4.1.2 Constants

`fio::seek_set = 0`

Constant for fseek that positions the file pointer relative to the beginning of the file by the given offset.

`fio::seek_cur = 1`

Constant for fseek that positions the file pointer relative to its current position by the given offset.

`fio::seek_end = 2`

Constant for fseek that positions the file pointer relative to the end of the file by the given offset.

`fio::df_magic = 0x12345678`

Magic number constant used to identify daslang binary file format. df_magic:uint const

4.1.3 Structures

`fio::df_header`

Obsolete header structure used by fsave and fload for binary serialization with type identification.

Fields

- **magic** : uint - Magic number constant used to identify and validate the file type.
- **size** : int - Total size in bytes of the serialized data following this header.

4.1.4 Handled structures

`fio::FStat`

`FStat.size() : uint64()`

Returns the size of the file in bytes.

`FStat.atime() : clock()`

Returns the last access time of the file as a clock value.

`FStat.ctime() : clock()`

Returns the creation time of the file as a clock value.

`FStat.mtime() : clock()`

Returns the last modification time of the file as a clock value.

`FStat.is_reg() : bool()`

Returns true if the file status indicates a regular file.

`FStat.`**`is_dir() : bool`**`()`

Returns true if the file status indicates a directory.

Properties

- **size** : uint64
- **atime** : *clock*
- **ctime** : *clock*
- **mtime** : *clock*
- **is_reg** : bool
- **is_dir** : bool

Fields

- **is_valid** : bool - *stat* and *fstat* return file information in this structure.

4.1.5 Handled types

`fio::`**`FILE`**

Opaque handle wrapping the platform-specific C FILE type used by all low-level file I/O functions.

4.1.6 File manipulation

- *fclose (file: FILE const?)*
- *feof (file: FILE const?) : bool*
- *fflush (file: FILE const?)*
- *fgets (file: FILE const?) : string*
- *fload (file: file; size: int; blk: block<(data:array<uint8>):void>)*
- *fload (f: file; var buf: auto(BufType)) : auto*
- *fmap (file: FILE const?; block: block<(array<uint8>#):void>)*
- *fopen (name: string; mode: string; blk: block<(f:file):void>) : auto*
- *fopen (name: string; mode: string) : FILE const?*
- *fprint (file: FILE const?; text: string)*
- *fread (f: file; buf: auto(BufType)) : auto*
- *fread (file: FILE const?) : string*
- *fread (f: file; buf: array<auto(BufType)>) : auto*
- *fread (f: file; blk: block<(data:string#):auto>) : auto*
- *fsave (f: file; buf: auto(BufType)) : auto*
- *fseek (file: FILE const?; offset: int64; mode: int) : int64*
- *fstat (file: FILE const?; stat: FStat) : bool*
- *fstat (f: file) : FStat*

- *fstderr () : FILE const?*
- *fstdin () : FILE const?*
- *fstdout () : FILE const?*
- *ftell (file: FILE const?) : int64*
- *fwrite (f: file; buf: array<auto(BufType)>) : auto*
- *fwrite (file: FILE const?; text: string)*
- *fwrite (f: file; buf: auto(BufType)) : auto*
- *getchar () : int*
- *remove (name: string) : bool*
- *rename (old_name: string; new_name: string) : bool*
- *stat (path: string) : FStat*
- *stat (file: string; stat: FStat) : bool*

`fio::`**`fclose`**(*file: FILE const?*)

Closes the given FILE pointer and releases its associated resources, equivalent to C fclose.

Arguments

- **file** : *FILE*? implicit

`fio::`**`feof(file: FILE const?) : bool`**()

Returns true if the end-of-file indicator has been set on the given FILE pointer, equivalent to C feof.

Arguments

- **file** : *FILE*? implicit

`fio::`**`fflush`**(*file: FILE const?*)

Flushes any buffered output data for the given FILE pointer to the underlying file, equivalent to C fflush.

Arguments

- **file** : *FILE*? implicit

`fio::`**`fgets(file: FILE const?) : string`**()

Reads and returns the next line as a string from the given FILE pointer, equivalent to C fgets.

Arguments

- **file** : *FILE*? implicit

fload

`fio::`**`fload`**(*file: file; size: int; blk: block<(data:array<uint8>):void>*)

Obsolete; loads binary data from a file into the provided buffer or passes it as an array of uint8 to a block.

Arguments

- **file** : *file*
- **size** : int

- **blk** : block<(data:array<uint8>):void>

fio::**fload(f: file; buf: auto(BufType)) : auto**()

---

fio::**fmap**(*file: FILE const?; block: block<(array<uint8>#):void>*)

Memory-maps the contents of the given FILE pointer and provides the data as an array of uint8 inside the block.

Arguments

- **file** : *FILE*? implicit
- **block** : block<(array<uint8>#):void> implicit

fopen

fio::**fopen(name: string; mode: string; blk: block<(f:file):void>) : auto**()

Opens the file at the given path with the specified mode string, returning a FILE pointer or invoking a block with a file handle.

Arguments

- **name** : string
- **mode** : string
- **blk** : block<(f: *file*):void>

fio::**fopen(name: string; mode: string) : FILE const?**()

---

fio::**fprint**(*file: FILE const?; text: string*)

Writes the given text string to the specified FILE pointer, equivalent to print but targeting a file.

Arguments

- **file** : *FILE*? implicit
- **text** : string implicit

fread

fio::**fread(f: file; buf: auto(BufType)) : auto**()

Reads data from a file into a buffer, an array, or returns the full contents as a string, with block-based overloads available.

Arguments

- **f** : *file*
- **buf** : auto(BufType) implicit

fio::**fread(file: FILE const?) : string**()

fio::**fread(f: file; buf: array<auto(BufType)>) : auto**()

fio::**fread(f: file; blk: block<(data:string#):auto>) : auto**()

---

fio::**fsave(f: file; buf: auto(BufType)) : auto**()

Obsolete; saves the provided buffer data to a file in binary format.

Arguments

- **f** : *file*
- **buf** : auto(BufType)

fio::**fseek(file: FILE const?; offset: int64; mode: int) : int64**()

Repositions the file pointer of the given FILE to the specified offset relative to the mode (seek_set, seek_cur, or seek_end) and returns the new position.

Arguments

- **file** : *FILE*? implicit
- **offset** : int64
- **mode** : int

fstat

fio::**fstat(file: FILE const?; stat: FStat) : bool**()

Retrieves file metadata such as size and timestamps into an FStat structure from a file handle, equivalent to C fstat.

Arguments

- **file** : *FILE*? implicit
- **stat** : *FStat* implicit

fio::**fstat(f: file) : FStat**()

---

fio::**fstderr() : FILE const?**()

Returns the FILE pointer corresponding to the standard error stream.

fio::**fstdin() : FILE const?**()

Returns the FILE pointer corresponding to the standard input stream.

fio::**fstdout() : FILE const?**()

Returns the FILE pointer corresponding to the standard output stream.

fio::**ftell(file: FILE const?) : int64**()

Returns the current byte offset of the file pointer for the given FILE, equivalent to C ftell.

Arguments

- **file** : *FILE*? implicit

fwrite

`fio::fwrite(f: file; buf: array<auto(BufType)>) : auto()`

Writes a string, typed buffer, or array of data to the specified file handle.

Arguments

- **f** : *file*
- **buf** : array<auto(BufType)> implicit

`fio::fwrite(`*file: FILE const?; text: string*`)`

`fio::fwrite(f: file; buf: auto(BufType)) : auto()`

`fio::getchar() : int()`

Reads and returns the next character from standard input as an integer, equivalent to C getchar.

`fio::remove(name: string) : bool()`

Deletes the file at the specified path and returns true if it was removed successfully.

Arguments

- **name** : string implicit

`fio::rename(old_name: string; new_name: string) : bool()`

Renames or moves a file from old_name to new_name and returns true on success.

Arguments

- **old_name** : string implicit
- **new_name** : string implicit

stat

`fio::stat(path: string) : FStat()`

Retrieves file metadata such as size and timestamps for the file at the given path, returning an FStat structure or populating one by reference.

Arguments

- **path** : string

`fio::stat(file: string; stat: FStat) : bool()`

4.1.7 Path manipulation

- *base_name (name: string) : string*
- *dir_name (name: string) : string*
- *get_full_file_name (path: string) : string*

`fio::`**`base_name(name: string) : string`**`()`

Extracts and returns the final component of a file path, equivalent to POSIX basename.

Arguments

- **name** : string implicit

`fio::`**`dir_name(name: string) : string`**`()`

Extracts and returns the directory component of a file path, equivalent to POSIX dirname.

Arguments

- **name** : string implicit

`fio::`**`get_full_file_name(path: string) : string`**`()`

Returns the fully resolved and normalized absolute path for the given file path string.

Arguments

- **path** : string implicit

4.1.8 Directory manipulation

- *chdir (path: string) : bool*
- *dir (path: string; blk: block<(filename:string):void>) : auto*
- *getcwd () : string*
- *mkdir (path: string) : bool*
- *mkdir_rec (path: string) : bool*

`fio::`**`chdir(path: string) : bool`**`()`

Changes the current working directory to the specified path and returns true on success.

Arguments

- **path** : string implicit

`fio::`**`dir(path: string; blk: block<(filename:string):void>) : auto`**`()`

Iterates over all entries in the directory at the given path, invoking the block with each filename.

Arguments

- **path** : string
- **blk** : block<(filename:string):void>

`fio::`**`getcwd() : string`**`()`

Returns the absolute path of the current working directory as a string.

fio::**mkdir(path: string) : bool**()

Creates a single directory at the specified path and returns true if it was created successfully.

Arguments

- **path** : string implicit

fio::**mkdir_rec(path: string) : bool**()

Recursively creates the directory at the specified path along with any missing parent directories, returning true on success.

Arguments

- **path** : string

4.1.9 OS specific routines

- *exit (exitCode: int)*
- *get_env_variable (var: string) : string*
- *has_env_variable (var: string) : bool*
- *popen (command: string; scope: block<(FILE const?):void>) : int*
- *popen_binary (command: string; scope: block<(FILE const?):void>) : int*
- *sanitize_command_line (var: string) : string*
- *sleep (msec: uint)*

fio::**exit**(*exitCode: int*)

Warning: This is unsafe operation.

Terminates the program immediately with the specified integer exit code, equivalent to C exit.

Arguments

- **exitCode** : int

fio::**get_env_variable(var: string) : string**()

Returns the string value of the environment variable with the given name, or an empty string if undefined.

Arguments

- **var** : string implicit

fio::**has_env_variable(var: string) : bool**()

Returns true if an environment variable with the given name is defined in the current process environment.

Arguments

- **var** : string implicit

fio::**popen(command: string; scope: block<(FILE const?):void>) : int**()

Warning: This is unsafe operation.

Opens a pipe to the given shell command, provides the resulting FILE pointer to the block, and returns the process exit code.

Arguments

- **command** : string implicit
- **scope** : block<(*FILE*?):void> implicit

fio::**popen_binary(command: string; scope: block<(FILE const?):void>) : int**()

Warning: This is unsafe operation.

Opens a pipe to the given shell command in binary mode, provides the resulting FILE pointer to the block, and returns the process exit code.

Arguments

- **command** : string implicit
- **scope** : block<(*FILE*?):void> implicit

fio::**sanitize_command_line(var: string) : string**()

Escapes and sanitizes a command-line argument string to prevent shell injection.

Arguments

- **var** : string implicit

fio::**sleep**(*msec: uint*)

Suspends execution of the current thread for the specified number of milliseconds.

Arguments

- **msec** : uint

4.1.10 Dynamic modules

- *register_dynamic_module (path: string; name: string) : void?*
- *register_dynamic_module (path: string; name: string; on_error: int) : void?*
- *register_native_path (mod_name: string; src: string; dst: string)*

register_dynamic_module

fio::**register_dynamic_module(path: string; name: string) : void?**()

Loads a shared library from the given path and registers it as a daslang module under the specified name, making it available for require.

Arguments

- **path** : string implicit
- **name** : string implicit

fio::**register_dynamic_module(path: string; name: string; on_error: int) : void?**()

---

fio::**register_native_path**(*mod_name: string; src: string; dst: string*)

Registers a path prefix mapping for a module, redirecting file resolution from the src prefix to the dst prefix.

Arguments

- **mod_name** : string implicit
- **src** : string implicit
- **dst** : string implicit

4.2 Network socket library

The NETWORK module implements networking facilities including HTTP client/server and low-level socket operations. It provides `Server` and `Client` classes with event-driven callbacks for handling connections, requests, and responses.

All functions and symbols are in "network" module, use require to get access to it.

```
require network
```

4.2.1 Handled structures

network::**NetworkServer**

Base implementation of the server.

4.2.2 Classes

network::**Server**

Fields

- **_server** : smart_ptr< *NetworkServer*> - Single-socket listener that manages one client connection at a time.

Server.**make_server_adapter**()

Creates a low-level server adapter bound to this `Server` instance.

Server.**init(port: int) : bool**()

Initializes the server on the specified port; returns `true` on success.

Arguments

- **port** : int

Server.**restore**(*shared_orphan: smart_ptr<NetworkServer>&*)

Restores the server with the given shared orphan network server pointer. This is necessary to re-establish the server state after reload of a script.

Arguments

- **shared_orphan** : smart_ptr< *NetworkServer*>&

Server.**save**(*shared_orphan: smart_ptr<NetworkServer>&*)

Saves the server state to a shared orphan network server pointer. This is necessary to re-establish the server state after reload of a script.

Arguments

- **shared_orphan** : smart_ptr< *NetworkServer*>&

Server.**has_session() : bool**()

Returns `true` if the server has an active client session.

Server.**is_open() : bool**()

Returns `true` if the server is open and accepting connections.

Server.**is_connected() : bool**()

Returns `true` if the server is currently connected to a client.

Server.**tick**()

Processes pending connections and incoming data; must be called periodically.

Server.**send(data: uint8?; size: int) : bool**()

Sends a data buffer to the connected client.

Arguments

- **data** : uint8?
- **size** : int

network::**Server() : Server**()

Constructs a new `Server` instance with default settings.

4.2.3 Low level NetworkServer IO

- *make_server (class: void?; info: StructInfo const?) : bool*
- *server_init (server: smart_ptr<NetworkServer>; port: int) : bool*
- *server_is_connected (server: smart_ptr<NetworkServer>) : bool*
- *server_is_open (server: smart_ptr<NetworkServer>) : bool*
- *server_restore (server: smart_ptr<NetworkServer>; class: void?; info: StructInfo const?)*
- *server_send (server: smart_ptr<NetworkServer>; data: uint8?; size: int) : bool*
- *server_tick (server: smart_ptr<NetworkServer>)*

`network::`**`make_server(class: void?; info: StructInfo const?) : bool`**`()`

Creates a new `Server` instance.

Arguments

- **class** : void? implicit
- **info** : *StructInfo*? implicit

`network::`**`server_init(server: smart_ptr<NetworkServer>; port: int) : bool`**`()`

Initializes the server to listen on the specified port.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit
- **port** : int

`network::`**`server_is_connected(server: smart_ptr<NetworkServer>) : bool`**`()`

Returns `true` if the server has an active client connection.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit

`network::`**`server_is_open(server: smart_ptr<NetworkServer>) : bool`**`()`

Returns `true` if the server is listening on its bound port.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit

`network::`**`server_restore`**`(`*server: smart_ptr<NetworkServer>; class: void?; info: StructInfo const?*`)`

Restores a server from an orphaned or interrupted state.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

`network::`**`server_send(server: smart_ptr<NetworkServer>; data: uint8?; size: int) : bool`**`()`

Sends data from the server to the connected client.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit
- **data** : uint8? implicit
- **size** : int

`network::`**`server_tick`**`(`*server: smart_ptr<NetworkServer>*`)`

Processes pending network I/O; must be called periodically for the server to function.

Arguments

- **server** : smart_ptr< *NetworkServer*> implicit

4.3 URI manipulation library based on UriParser

The URIPARSER module provides URI parsing and manipulation based on the uriparser library. It supports parsing URI strings into components (scheme, host, path, query, fragment), normalization, resolution of relative URIs, and GUID generation.

All functions and symbols are in "uriparser" module, use require to get access to it.

```
require uriparser
```

4.3.1 Handled structures

`uriparser::`**`UriTextRangeA`**

Range of text in the URI.

`uriparser::`**`UriIp4Struct`**

IPv4 address portion of the URI.

Fields

- **data** : uint8[4] - IPv4 address data.

`uriparser::`**`UriIp6Struct`**

IPv6 address portion of the URI.

Fields

- **data** : uint8[16] - IPv6 address data.

`uriparser::`**`UriHostDataA`**

Host data portion of the URI (IPv4 or IPv6, or some future data).

Fields

- **ip4** : *UriIp4Struct*? - IPv4 address data.
- **ip6** : *UriIp6Struct*? - IPv6 address data.
- **ipFuture** : *UriTextRangeA* - Future host address data.

`uriparser::`**`UriPathSegmentStructA`**

Part of the path portion of the URI.

Fields

- **text** : *UriTextRangeA* - Text of the path segment.
- **next** : *UriPathSegmentStructA*? - Next path segment, or null if this is the last segment.

`uriparser::`**`UriUriA`**

URI base class, contains all URI data.

Fields

- **scheme** : *UriTextRangeA* - Scheme of the URI.
- **userInfo** : *UriTextRangeA* - User information.
- **hostText** : *UriTextRangeA* - Host text.
- **hostData** : *UriHostDataA* - Host data portion of the URI (IPv4 or IPv6, or some future data).
- **portText** : *UriTextRangeA* - Port text.
- **pathHead** : *UriPathSegmentStructA*? - Head of the path.
- **pathTail** : *UriPathSegmentStructA*? - Tail of the path.
- **query** : *UriTextRangeA* - Query portion of the URI.
- **fragment** : *UriTextRangeA* - Fragment portion of the URI.
- **absolutePath** : int - Whether the path is absolute.
- **owner** : int - Whether the URI is owned by the parser.

`uriparser::`**`Uri`**

`Uri.`**`empty() : bool`**`()`

Returns `true` if the `Uri` object contains no URI data.

`Uri.`**`size() : int`**`()`

Returns the string length of the URI.

`Uri.`**`status() : int`**`()`

Returns the parse status code of the `Uri` object.

Properties

- **empty** : bool
- **size** : int
- **status** : int

Fields

- **uri** : *UriUriA* - URI implementation.

4.3.2 Initialization and finalization

- *Uri (arg0: string) : Uri*
- *Uri () : Uri*
- *clone (dest: Uri; src: Uri)*
- *finalize (uri: Uri)*
- *using (arg0: string; arg1: block<(Uri#):void>)*
- *using (arg0: block<(Uri#):void>)*

Uri

`uriparser::Uri(arg0: string) : Uri()`

Constructs a new `Uri` object by parsing the given URI string.

Arguments

- **arg0** : string implicit

`uriparser::Uri() : Uri()`

`uriparser::clone(`*dest: Uri; src: Uri*`)`

Creates a deep copy of the given `Uri` object.

Arguments

- **dest** : *Uri* implicit
- **src** : *Uri* implicit

`uriparser::finalize(`*uri: Uri*`)`

Releases all resources held by the `Uri` object.

Arguments

- **uri** : *Uri* implicit

using

`uriparser::using(`*arg0: string; arg1: block<(Uri#):void>*`)`

Creates a scoped `Uri` variable that is automatically finalized at end of block.

Arguments

- **arg0** : string implicit
- **arg1** : block<(*Uri#*):void> implicit

`uriparser::using(`*arg0: block<(Uri#):void>*`)`

4.3.3 Escape and unescape

- *escape_uri (uriStr: string; spaceToPlus: bool; normalizeBreaks: bool) : string*
- *unescape_uri (uriStr: string) : string*

`uriparser::`**`escape_uri(uriStr: string; spaceToPlus: bool; normalizeBreaks: bool) : string`**`()`

Percent-encodes reserved and special characters in the URI string.

Arguments

- **uriStr** : string implicit
- **spaceToPlus** : bool
- **normalizeBreaks** : bool

`uriparser::`**`unescape_uri(uriStr: string) : string`**`()`

Decodes percent-encoded characters in the URI string.

Arguments

- **uriStr** : string implicit

4.3.4 Uri manipulations

- *add_base_uri (base: Uri; relative: Uri) : Uri*
- *normalize (uri: Uri) : bool*
- *normalize_uri (uriStr: string) : string*
- *remove_base_uri (base: Uri; relative: Uri) : Uri*
- *string (uri: Uri) : string*
- *string (range: UriTextRangeA) : string*
- *strip_uri (uri: Uri; query: bool; fragment: bool) : Uri*
- *uri_for_each_query_kv (uri: Uri; block: block<(string#;string#):void>)*

`uriparser::`**`add_base_uri(base: Uri; relative: Uri) : Uri`**`()`

Resolves a relative URI against a base URI, producing an absolute URI.

Arguments

- **base** : *Uri* implicit
- **relative** : *Uri* implicit

`uriparser::`**`normalize(uri: Uri) : bool`**`()`

Normalizes a `Uri` in place, removing redundant `/`, `.`, and `..` path segments.

Arguments

- **uri** : *Uri* implicit

uriparser::**normalize_uri(uriStr: string) : string**()

Returns a normalized copy of the URI string with redundant `/`, `.`, and `..` segments removed.

Arguments

- **uriStr** : string implicit

uriparser::**remove_base_uri(base: Uri; relative: Uri) : Uri**()

Computes a relative URI by removing the base URI prefix from an absolute URI.

Arguments

- **base** : *Uri* implicit
- **relative** : *Uri* implicit

string

uriparser::**string(uri: Uri) : string**()

Converts a `Uri` object to its string representation.

Arguments

- **uri** : *Uri* implicit

uriparser::**string(range: UriTextRangeA) : string**()

---

uriparser::**strip_uri(uri: Uri; query: bool; fragment: bool) : Uri**()

Removes the query string and fragment from the URI.

Arguments

- **uri** : *Uri* implicit
- **query** : bool
- **fragment** : bool

uriparser::**uri_for_each_query_kv**(*uri: Uri; block: block<(string#;string#):void>*)

Iterates over each key-value pair in the URI's query string, invoking a block for each.

Arguments

- **uri** : *Uri* implicit
- **block** : block<(string#;string#):void> implicit

4.3.5 File name conversions

- *file_name_to_uri (uriStr: string) : string*
- *to_file_name (uri: Uri) : string*
- *to_unix_file_name (uri: Uri) : string*
- *to_windows_file_name (uri: Uri) : string*
- *unix_file_name_to_uri (uriStr: string) : string*
- *uri_from_file_name (filename: string) : Uri*
- *uri_from_unix_file_name (filename: string) : Uri*
- *uri_from_windows_file_name (filename: string) : Uri*
- *uri_to_file_name (uriStr: string) : string*
- *uri_to_unix_file_name (uriStr: string) : string*
- *uri_to_windows_file_name (uriStr: string) : string*
- *windows_file_name_to_uri (uriStr: string) : string*

`uriparser::`**`file_name_to_uri(uriStr: string) : string`**`()`

Converts a platform-native file path to a `file://` URI string.

Arguments

- **uriStr** : string implicit

`uriparser::`**`to_file_name(uri: Uri) : string`**`()`

Converts a `Uri` to a platform-native file path.

Arguments

- **uri** : *Uri* implicit

`uriparser::`**`to_unix_file_name(uri: Uri) : string`**`()`

Converts a `Uri` to a Unix-style file path.

Arguments

- **uri** : *Uri* implicit

`uriparser::`**`to_windows_file_name(uri: Uri) : string`**`()`

Converts a `Uri` to a Windows-style file path.

Arguments

- **uri** : *Uri* implicit

`uriparser::`**`unix_file_name_to_uri(uriStr: string) : string`**`()`

Converts a Unix-style file path to a `file://` URI string.

Arguments

- **uriStr** : string implicit

`uriparser::`**`uri_from_file_name(filename: string) : Uri`**`()`

Converts a platform-native file path to a `file://` URI string.

Arguments

- **filename** : string implicit

`uriparser::`**`uri_from_unix_file_name(filename: string) : Uri`**`()`

Converts a Unix-style file path to a `file://` URI string.

Arguments

- **filename** : string implicit

`uriparser::`**`uri_from_windows_file_name(filename: string) : Uri`**`()`

Converts a Windows-style file path to a `file://` URI string.

Arguments

- **filename** : string implicit

`uriparser::`**`uri_to_file_name(uriStr: string) : string`**`()`

Converts a URI string to a platform-native file path.

Arguments

- **uriStr** : string implicit

`uriparser::`**`uri_to_unix_file_name(uriStr: string) : string`**`()`

Converts a URI string to a Unix-style file path.

Arguments

- **uriStr** : string implicit

`uriparser::`**`uri_to_windows_file_name(uriStr: string) : string`**`()`

Converts a URI string to a Windows-style file path.

Arguments

- **uriStr** : string implicit

`uriparser::`**`windows_file_name_to_uri(uriStr: string) : string`**`()`

Converts a Windows-style file path to a `file://` URI string.

Arguments

- **uriStr** : string implicit

4.3.6 GUID

- *make_new_guid () : string*

`uriparser::`**`make_new_guid() : string`**`()`

Generates a new random GUID/UUID string.

4.4 Boost package for the URI parser

The URIPARSER_BOOST module extends URI handling with convenience functions for common operations like building URIs from components, extracting query parameters, and resolving relative paths.

All functions and symbols are in "uriparser_boost" module, use require to get access to it.

```
require daslib/uriparser_boost
```

4.4.1 Split and compose

- *uri_compose (scheme: string; userInfo: string; hostText: string; portText: string; path: string; query: string; fragment: string) : Uri*
- *uri_compose_query (query: table<string, string>) : string*
- *uri_compose_query_in_order (query: table<string, string>) : string*
- *uri_split_full_path (uri: Uri) : array<string>*

`uriparser_boost::`**`uri_compose(scheme: string; userInfo: string; hostText: string; portText: string; path`**

Compose a URI from its components.

Arguments

- **scheme** : string
- **userInfo** : string
- **hostText** : string
- **portText** : string
- **path** : string
- **query** : string
- **fragment** : string

`uriparser_boost::`**`uri_compose_query(query: table<string, string>) : string`**`()`

Compose a query string from a table of key-value pairs.

Arguments

- **query** : table<string;string>

`uriparser_boost::`**`uri_compose_query_in_order(query: table<string, string>) : string`**`()`

Compose a query string from a table of key-value pairs, in the sorted order.

Arguments

- **query** : table<string;string>

`uriparser_boost::`**`uri_split_full_path(uri: Uri) : array<string>`**`()`

Split the full path of a URI into its components.

Arguments

- **uri** : *Uri* implicit

4.4.2 Component accessors

- *fragment (uri: Uri) : string*
- *host (uri: Uri) : string*
- *path (uri: Uri) : string*
- *port (uri: Uri) : string*
- *query (uri: Uri) : string*
- *scheme (uri: Uri) : string*
- *user_info (uri: Uri) : string*

`uriparser_boost::`**`fragment(uri: Uri) : string`**`()`

Return the fragment of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`host(uri: Uri) : string`**`()`

Return the host of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`path(uri: Uri) : string`**`()`

Return the path of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`port(uri: Uri) : string`**`()`

Return the port of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`query(uri: Uri) : string`**`()`

Return the query of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`scheme(uri: Uri) : string`**`()`

Returns the scheme of a URI.

Arguments

- **uri** : *Uri* implicit

`uriparser_boost::`**`user_info(uri: Uri) : string`**`()`

Return the user info of a URI.

Arguments

- **uri** : *Uri* implicit

4.5 General purpose serialization

The ARCHIVE module implements general-purpose serialization infrastructure. It provides the `Archive` type and `serialize` functions for reading and writing binary data. Custom types are supported by implementing `serialize` for each type.

All functions and symbols are in "archive" module, use require to get access to it.

```
require daslib/archive
```

To correctly support serialization of the specific type, you need to define and implement `serialize` method for it. For example this is how DECS implements component serialization:

```
def public serialize ( var arch:Archive; var src:Component )
    arch |> serialize(src.name)
    arch |> serialize(src.hash)
    arch |> serialize(src.stride)
    arch |> serialize(src.info)
    invoke(src.info.serializer, arch, src.data)
```

Example:

```
require daslib/archive

    struct Foo {
        a : float
        b : string
    }

    [export]
    def main() {
        var original = Foo(a = 3.14, b = "hello")
        var data <- mem_archive_save(original)
        var loaded : Foo
        data |> mem_archive_load(loaded)
        delete data
        print("a = {loaded.a}, b = {loaded.b}\n")
    }
    // output:
    // a = 3.14, b = hello
```

4.5.1 Structures

`archive::`**`Archive`**

Archive is a combination of serialization stream, and state (version, and reading status).

Fields

- **version** : uint - Version of the archive format.
- **reading** : bool - True if the archive is for reading, false for writing.
- **stream** : *Serializer*? - Serialization stream.

4.5.2 Classes

`archive::`**`Serializer`**

Base class for serializers.

`archive::`**`MemSerializer : Serializer`**

This serializer stores data in memory (in the array<uint8>) internal data buffer current reading offset last error code

`MemSerializer.`**`write(bytes: void?; size: int) : bool`**`()`

Appends bytes at the end of the data.

Arguments

- **bytes** : void? implicit
- **size** : int

`MemSerializer.`**`read(bytes: void?; size: int) : bool`**`()`

Reads bytes from data, advances the reading position.

Arguments

- **bytes** : void? implicit
- **size** : int

`MemSerializer.`**`error`**`(`*code: string*`)`

Sets the last error code.

Arguments

- **code** : string

`MemSerializer.`**`OK() : bool`**`()`

Implements 'OK' method, which returns true if the serializer is in a valid state.

`MemSerializer.`**`extractData() : array<uint8>`**`()`

Extract the data from the serializer.

`MemSerializer.`**`getCopyOfData() : array<uint8>`**`()`

Returns copy of the data from the serializer.

`MemSerializer.`**`getLastError() : string`**`()`

Returns last serialization error.

`archive::`**`MemSerializer() : MemSerializer`**`()`

Initialize the serializer for reading or writing.

`archive::`**`MemSerializer(from: array<uint8>) : MemSerializer`**`()`

Initialize the serializer for reading from the given data.

Arguments

- **from** : array<uint8>

4.5.3 Serialization

- *read_raw (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: float3x4)*
- *serialize (var arch: Archive; var value: string&)*
- *serialize (var arch: Archive; var value: float3x3)*
- *serialize (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: float4x4)*
- *serialize (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: table<auto(KT), auto(VT)>) : auto*
- *serialize (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: auto(TT)&) : auto*
- *serialize (var arch: Archive; var value: auto(TT)[]) : auto*
- *serialize (var arch: Archive; var value: array<auto(TT)>) : auto*
- *serialize (var arch: Archive; var value: auto(TT)?) : auto*
- *serialize_raw (var arch: Archive; var value: auto(TT)&) : auto*
- *write_raw (var arch: Archive; var value: auto(TT)&) : auto*

`archive::`**`read_raw(arch: Archive; value: auto(TT)&) : auto`**`()`

Read raw data (straight up bytes for raw pod)

Arguments

- **arch** : *Archive*
- **value** : auto(TT)&

serialize

`archive::`**`serialize`**`(`*arch: Archive; value: float3x4*`)`

Serializes float3x4 matrix

Arguments

- **arch** : *Archive*
- **value** : *float3x4*

`archive::`**`serialize`**`(`*arch: Archive; value: string&*`)`

`archive::`**`serialize`**`(`*arch: Archive; value: float3x3*`)`

`archive::`**`serialize(arch: Archive; value: auto(TT)&) : auto()`**

`archive::`**`serialize`**`(`*arch: Archive; value: float4x4*`)`

`archive::`**`serialize(arch: Archive; value: auto(TT)&) : auto()`**

`archive::`**`serialize(arch: Archive; value: table<auto(KT), auto(VT)>) : auto()`**

`archive::`**`serialize(arch: Archive; value: auto(TT)&) : auto()`**

`archive::`**`serialize(arch: Archive; value: auto(TT)&) : auto()`**

`archive::`**`serialize(arch: Archive; value: auto(TT)&) : auto()`**

`archive::`**`serialize(arch: Archive; value: auto(TT)[]) : auto()`**

`archive::`**`serialize(arch: Archive; value: array<auto(TT)>) : auto()`**

`archive::`**`serialize(arch: Archive; value: auto(TT)?) : auto()`**

`archive::`**`serialize_raw(arch: Archive; value: auto(TT)&) : auto()`**

Serialize raw data (straight up bytes for raw pod)

Arguments

- **arch** : *Archive*
- **value** : auto(TT)&

`archive::`**`write_raw(arch: Archive; value: auto(TT)&) : auto()`**

Write raw data (straight up bytes for raw pod)

Arguments

- **arch** : *Archive*
- **value** : auto(TT)&

4.5.4 Memory archive

- *mem_archive_load (var data: array<uint8>; var t: auto&; canfail: bool = false) : bool*
- *mem_archive_save (var t: auto&) : auto*

`archive::`**`mem_archive_load(data: array<uint8>; t: auto&; canfail: bool = false) : bool`**`()`

Loads the object from a memory archive. *data* is the array<uint8> with the serialized data, returned from *mem_archive_save*.

Arguments

- **data** : array<uint8>
- **t** : auto&
- **canfail** : bool

`archive::`**`mem_archive_save(t: auto&) : auto`**`()`

Saves the object to a memory archive. Result is array<uint8> with the serialized data.

Arguments

- **t** : auto&

CHAPTER

FIVE

DATA STRUCTURES

Container extensions, sorting, hash tables, linked lists, and specialized arrays.

5.1 Boost package for array manipulation

The ARRAY_BOOST module extends array operations with temporary array views over fixed-size arrays and C++ handled vectors, emptiness checks, sub-array views, and arithmetic operators on fixed-size arrays.

All functions and symbols are in "array_boost" module, use require to get access to it.

```
require daslib/array_boost
```

5.1.1 Temporary arrays

- *temp_array (arr: auto const implicit ==const) : auto*
- *temp_array (var arr: auto implicit ==const) : auto*
- *temp_array (data: auto? ==const; lenA: int; a: auto(TT)) : array<TT>*
- *temp_array (var data: auto? ==const; lenA: int; a: auto(TT)) : array<TT>*

temp_array

array_boost::**temp_array(arr: auto const implicit ==const) : auto**()

Warning: This is unsafe operation.

Creates temporary array from the given object. Important requirements are:

- object memory is linear
- each element follows the next one directly, with the stride equal to size of the element
- object memory does not change within the lifetime of the returned array

Arguments

- **arr** : auto implicit!

array_boost::**temp_array(arr: auto implicit ==const) : auto**()

array_boost::**temp_array(data: auto? ==const; lenA: int; a: auto(TT)) : array<TT>**()

array_boost::**temp_array(data: auto? ==const; lenA: int; a: auto(TT)) : array<TT>**()

5.1.2 Empty check

- *empty (v: auto(VecT)) : auto*

array_boost::**empty(v: auto(VecT)) : auto**()

returns true if 'v' has 0 elements. this also implies that *length(v)* is defined.

Arguments

- **v** : auto(VecT)

5.1.3 Sub-array view

- *array_view (var bytes: array<auto(TT)>; offset: int; length: int; blk: block<(var view:array<TT>#):void>) : auto*
- *array_view (bytes: array<auto(TT)>; offset: int; length: int; blk: block<(view:array<TT>#):void>) : auto*

array_view

array_boost::**array_view(bytes: array<auto(TT)>; offset: int; length: int; blk: block<(var view:array<TT**

creates a view of the array, which is a temporary array that is valid only within the block

Arguments

- **bytes** : array<auto(TT)>!
- **offset** : int
- **length** : int
- **blk** : block<(view:array<TT>#):void>

array_boost::**array_view(bytes: array<auto(TT)>; offset: int; length: int; blk: block<(view:array<TT>#):**

5.2 Boost package for the builtin sort

The SORT_BOOST module provides the `qsort` macro that uniformly sorts built-in arrays, dynamic arrays, and C++ handled vectors using the same syntax. It automatically wraps handled types in `temp_array` as needed.

All functions and symbols are in "sort_boost" module, use require to get access to it.

```
require daslib/sort_boost
```

5.2.1 Call macros

`sort_boost::`**`qsort`**

Implements *qsort* macro. It's *qsort(value,block)*. For the regular array<> or dim it's replaced with *sort(value,block)*. For the handled types like das`vector its replaced with *sort(temp_array(value),block)*.

5.3 Flat hash table

The FLAT_HASH_TABLE module implements a flat (open addressing) hash table. It stores all entries in a single contiguous array, providing cache-friendly access patterns and good performance for small to medium-sized tables.

All functions and symbols are in "flat_hash_table" module, use require to get access to it.

```
require daslib/flat_hash_table
```

Example:

```
require daslib/flat_hash_table public

    typedef IntMap = TFlatHashTable<int; string>

    [export]
    def main() {
        var m <- IntMap()
        m[1] = "one"
        m[2] = "two"
        m[3] = "three"
        print("length = {m.data_length}\n")
        print("m[2] = {m[2]}\n")
        m.clear()
        print("after clear: {m.data_length}\n")
    }
    // output:
    // length = 3
    // m[2] = two
    // after clear: 0
```

5.3.1 Type macros

`flat_hash_table::`**`type macro TFlatHashTable`**

Arguments

- **ValueType** (Type)
- **hashFunctionName** (String = "hash")

5.4 Cuckoo hash table

The CUCKOO_HASH_TABLE module implements a cuckoo hash table data structure. Cuckoo hashing provides worst-case O(1) lookup time by using multiple hash functions and displacing existing entries on collision.

All functions and symbols are in “cuckoo_hash_table” module, use require to get access to it.

```
require daslib/cuckoo_hash_table
```

5.4.1 Type macros

cuckoo_hash_table::**type macro TCuckooHashTable**

Arguments

- **ValueType** (Type)
- **hashFunction0Name** (String = “hash0”)
- **hashFunction1Name** (String = “hash_extra”)

5.4.2 Hash functions

- *hash0 (data: auto) : auto*
- *hash_extra (data: auto) : auto*

cuckoo_hash_table::**hash0(data: auto) : auto**()

this hash function converts and workhorse key to a 64 bit hash

Arguments

- **data** : auto

cuckoo_hash_table::**hash_extra(data: auto) : auto**()

Returns a secondary hash derived from the upper 32 bits of the primary hash, used for cuckoo hashing.

Arguments

- **data** : auto

5.5 Intrusive linked list

The LINKED_LIST module implements intrusive linked list data structures. Elements contain embedded next/prev pointers, avoiding separate node allocations. Useful for implementing queues, work lists, and other dynamic collections with O(1) insertion and removal.

All functions and symbols are in “linked_list” module, use require to get access to it.

```
require daslib/linked_list
```

5.5.1 Constants

TLinkedList = %spoof_template

Spoof template that generates a doubly-linked list class for a given element type. Usage: `apply_template(TLinkedList, "MyList", "MyElement")`

5.6 SOA (Structure of Arrays) transformation

The SOA (Structure of Arrays) module transforms array-of-structures data layouts into structure-of-arrays layouts for better cache performance. It provides macros that generate parallel arrays for each field of a structure, enabling SIMD-friendly data access patterns.

See tutorial_soa for a hands-on tutorial.

All functions and symbols are in "soa" module, use require to get access to it.

```
require daslib/soa
```

5.6.1 Structures

soa::**SOA_INDEX**

Proxy type returned by the `[]` operator on an SOA structure. Field access on this proxy is rewritten by *SoaCallMacro* to index into the correct column array.

5.6.2 Function annotations

soa::**SoaCallMacro**

Rewrites `soa[index].field` into `soa.field[index]` at compile time. This is the core of the SOA access pattern — it transforms AOS-style element access into column-wise array indexing for better cache locality.

5.6.3 Structure macros

soa::**soa**

Generates a Structure-of-Arrays layout from a regular struct. For a struct `Foo` with fields `x : float` and `y : float`, generates:

- ``Foo`SOA`` — struct where every field is `array<FieldType>`
- `operator []` returning `SOA_INDEX` proxy
- `length`, `capacity`, `push`, `push_clone`, `emplace`, `erase`, `pop`, `clear`, `resize`, `reserve`, `swap`, `from_array`, `to_array` functions

5.6.4 SOA field access

- *SOA_INDEX. (src: SOA_INDEX; field: string)*

`SOA_INDEX.`(*src: SOA_INDEX; field: string*)

Field access operator for SOA_INDEX; rewritten by SoaCallMacro to convert soa[index].field into soa.field[index].

Arguments

- **src** : *SOA_INDEX*
- **field** : string

5.7 Packed boolean array

The BOOL_ARRAY module provides a compact boolean array implementation using bit-packing. Each boolean value uses a single bit instead of a byte, providing an 8x memory reduction compared to `array<bool>`.

All functions and symbols are in "bool_array" module, use require to get access to it.

```
require daslib/bool_array
```

5.7.1 Structures

`bool_array::`**`BoolArray`**

A dynamic array of booleans, stored as bits.

`BoolArray.`**`finalize`**`()`

Releases the memory used by the BoolArray.

`BoolArray.`**`[](index: int) : bool`**`()`

Get the boolean value at the given index.

Arguments

- **index** : int

`BoolArray.`**`[]=`**(*index: int; value: bool*)

Set the boolean value at the given index.

Arguments

- **index** : int
- **value** : bool

`BoolArray.`**`[]^^=`**(*index: int; value: bool*)

Perform XOR operation on the boolean value at the given index.

Arguments

- **index** : int
- **value** : bool

`BoolArray.[]&&=`(*index: int; value: bool*)

Perform AND operation on the boolean value at the given index.

Arguments

- **index** : int
- **value** : bool

`BoolArray.[]||=`(*index: int; value: bool*)

Perform OR operation on the boolean value at the given index.

Arguments

- **index** : int
- **value** : bool

`BoolArray.clear()`

Clear the BoolArray.

`BoolArray.reserve`(*capacity: int*)

Reserve capacity for the BoolArray.

Arguments

- **capacity** : int

`BoolArray.resize`(*newSize: int*)

Resize the BoolArray to the new size.

Arguments

- **newSize** : int

`BoolArray.push`(*value: bool*)

Push a new boolean value to the end of the BoolArray.

Arguments

- **value** : bool

`BoolArray.push`(*value: bool; at: int*)

Push a new boolean value at the given index in the BoolArray.

Arguments

- **value** : bool
- **at** : int

`BoolArray.pop() : bool()`

Pop the last boolean value from the BoolArray and return it.

`BoolArray.length() : int()`

Get the length of the BoolArray.

`BoolArray.`**`erase`**`(`*index: int*`)`

Erase the boolean value at the given index from the BoolArray.

Arguments

- **index** : int

`BoolArray.`**`insert`**`(`*index: int; value: bool*`)`

Insert a boolean value at the given index in the BoolArray.

Arguments

- **index** : int
- **value** : bool

`BoolArray.`**`to_string() : string`**`()`

Convert the BoolArray to a string representation.

`BoolArray.`**`data_pointer() : uint?`**`()`

> **Warning:** This is unsafe operation.

Get the data pointer of the BoolArray.

`BoolArray.`**`data_pointer() : uint const?`**`()`

> **Warning:** This is unsafe operation.

Get the data pointer of the BoolArray.

5.7.2 Iteration

- *each (self: BoolArray) : iterator<bool>*

`bool_array::`**`each(self: BoolArray) : iterator<bool>`**`()`

Returns an iterator over all boolean values in the BoolArray.

Arguments

- **self** : *BoolArray*

CHAPTER

SIX

ALGORITHMS AND FUNCTIONAL

Generic algorithms, functional programming utilities, LINQ-style queries, and pattern matching.

6.1 Miscellaneous algorithms

The ALGORITHM module provides array and collection manipulation algorithms including sorting, searching, set operations, element removal, and more.

Key features:

- **Search**: *lower_bound*, *upper_bound*, *binary_search*, *equal_range*
- **Array manipulation**: *unique*, *sort_unique*, *reverse*, *combine*, *fill*, *rotate*, *erase_all*, *min_element*, *max_element*, *is_sorted*, *topological_sort*
- **Set operations** (on tables): *intersection*, *union*, *difference*, *symmetric_difference*, *identical*, *is_subset*

Most functions also support fixed-size arrays via *[expect_any_array]* overloads.

See tutorial_algorithm for a hands-on tutorial.

All functions and symbols are in "algorithm" module, use require to get access to it.

```
require daslib/algorithm
```

Example:

```
require daslib/algorithm

    [export]
    def main() {
        var arr <- [3, 1, 4, 1, 5, 9, 2, 6, 5]
        sort_unique(arr)
        print("sort_unique: {arr}\n")
        print("has 4: {binary_search(arr, 4)}\n")
        print("has 7: {binary_search(arr, 7)}\n")
        print("lower_bound(4): {lower_bound(arr, 4)}\n")
        print("upper_bound(4): {upper_bound(arr, 4)}\n")
        let er = equal_range(arr, 5)
        print("equal_range(5): {er}\n")
        print("min index: {min_element(arr)}\n")
        print("is_sorted: {is_sorted(arr)}\n")
    }
```

6.1.1 Search

- *binary_search (a: auto; f: int; last: int; val: auto) : auto*
- *binary_search (a: array<auto(TT)>; val: TT) : auto*
- *binary_search (a: auto; f: int; last: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*
- *binary_search (a: auto; val: auto) : auto*
- *binary_search (a: array<auto(TT)>; val: TT; less: block<(a:TT;b:TT):bool>) : auto*
- *binary_search (a: array<auto(TT)>; f: int; last: int; val: TT) : auto*
- *binary_search (a: array<auto(TT)>; f: int; last: int; val: TT; less: block<(a:TT;b:TT):bool>) : auto*
- *binary_search (a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*
- *equal_range (a: array<auto(TT)>; f: int; l: int; val: TT) : auto*
- *equal_range (a: array<auto(TT)>; val: TT) : auto*
- *lower_bound (a: array<auto(TT)>; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto*
- *lower_bound (a: array<auto(TT)>; val: TT) : auto*
- *lower_bound (a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*
- *lower_bound (a: array<auto(TT)>; f: int; l: int; val: TT) : auto*
- *lower_bound (a: array<auto(TT)>; f: int; l: int; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto*
- *lower_bound (a: auto; f: int; l: int; val: auto) : auto*
- *lower_bound (a: auto; val: auto) : auto*
- *lower_bound (a: auto; f: int; l: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*
- *upper_bound (a: array<auto(TT)>; f: int; l: int; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto*
- *upper_bound (a: array<auto(TT)>; val: TT) : auto*
- *upper_bound (a: auto; f: int; l: int; val: auto) : auto*
- *upper_bound (a: array<auto(TT)>; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto*
- *upper_bound (a: auto; val: auto) : auto*
- *upper_bound (a: array<auto(TT)>; f: int; l: int; val: TT) : auto*
- *upper_bound (a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*
- *upper_bound (a: auto; f: int; l: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto*

binary_search

`algorithm::`**`binary_search(a: auto; f: int; last: int; val: auto) : auto`**`()`

Returns true if val appears within the range [f, last).

Arguments

- **a** : auto
- **f** : int
- **last** : int

- **val** : auto

algorithm::**binary_search(a: array<auto(TT)>; val: TT) : auto**()

algorithm::**binary_search(a: auto; f: int; last: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : au**

algorithm::**binary_search(a: auto; val: auto) : auto**()

algorithm::**binary_search(a: array<auto(TT)>; val: TT; less: block<(a:TT;b:TT):bool>) : auto**()

algorithm::**binary_search(a: array<auto(TT)>; f: int; last: int; val: TT) : auto**()

algorithm::**binary_search(a: array<auto(TT)>; f: int; last: int; val: TT; less: block<(a:TT;b:TT):bool>)**

algorithm::**binary_search(a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto**()

---

equal_range

algorithm::**equal_range(a: array<auto(TT)>; f: int; l: int; val: TT) : auto**()

Returns a pair of indices [lower, upper) bounding the range of elements equal to val within [f, l). The array must be sorted.

Arguments

- **a** : array<auto(TT)>
- **f** : int
- **l** : int
- **val** : TT

algorithm::**equal_range(a: array<auto(TT)>; val: TT) : auto**()

---

lower_bound

algorithm::**lower_bound(a: array<auto(TT)>; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto**()

Returns the index of the first element in the array for which less returns false, or length(a) if no such element is found.

Arguments

- **a** : array<auto(TT)>
- **value** : auto(QQ)
- **less** : block<(a:TT;b:QQ):bool>

algorithm::**lower_bound(a: array<auto(TT)>; val: TT) : auto**()

algorithm::**lower_bound(a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto**()

algorithm::**lower_bound(a: array<auto(TT)>; f: int; l: int; val: TT) : auto**()

algorithm::**lower_bound(a: array<auto(TT)>; f: int; l: int; value: auto(QQ); less: block<(a:TT;b:QQ):boo** `algorithm::lower_bound(a: auto; f: int; l: int; val: auto) : auto()`

`algorithm::lower_bound(a: auto; val: auto) : auto()`

`algorithm::lower_bound(a: auto; f: int; l: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto()`

---

upper_bound

`algorithm::upper_bound(a: array<auto(TT)>; f: int; l: int; value: auto(QQ); less: block<(a:TT;b:QQ):boo`

Returns the index of the first element in the range [f, l) for which less(val, element) returns true, or l if no such element is found.

Arguments

- **a** : array<auto(TT)>
- **f** : int
- **l** : int
- **value** : auto(QQ)
- **less** : block<(a:TT;b:QQ):bool>

`algorithm::upper_bound(a: array<auto(TT)>; val: TT) : auto()`

`algorithm::upper_bound(a: auto; f: int; l: int; val: auto) : auto()`

`algorithm::upper_bound(a: array<auto(TT)>; value: auto(QQ); less: block<(a:TT;b:QQ):bool>) : auto()`

`algorithm::upper_bound(a: auto; val: auto) : auto()`

`algorithm::upper_bound(a: array<auto(TT)>; f: int; l: int; val: TT) : auto()`

`algorithm::upper_bound(a: auto; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto()`

`algorithm::upper_bound(a: auto; f: int; l: int; val: auto(TT); less: block<(a:TT;b:TT):bool>) : auto()`

6.1.2 Array manipulation

- *combine (a: array<auto(TT)>; b: array<auto(TT)>) : auto*
- *combine (a: auto; b: auto) : auto*
- *erase_all (var arr: auto; value: auto) : auto*
- *fill (var a: array<auto(TT)>; value: TT) : auto*
- *fill (var a: auto; value: auto) : auto*
- *is_sorted (a: auto) : bool*
- *is_sorted (a: array<auto(TT)>) : bool*
- *is_sorted (a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : bool*
- *max_element (a: auto) : int*
- *max_element (a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : int*

- *max_element (a: array<auto(TT)>) : int*
- *min_element (a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : int*
- *min_element (a: auto) : int*
- *min_element (a: array<auto(TT)>) : int*
- *reverse (var a: auto) : auto*
- *reverse (var a: array<auto>) : auto*
- *rotate (var a: array<auto>; mid: int) : auto*
- *rotate (var a: auto; mid: int) : auto*
- *sort_unique (var a: array<auto(TT)>) : auto*
- *topological_sort (nodes: array<auto(Node)>) : auto*
- *unique (var a: array<auto(TT)>) : auto*

combine

`algorithm::`**`combine(a: array<auto(TT)>; b: array<auto(TT)>) : auto`**`()`

Returns a new array containing elements from a followed by b.

Arguments

- **a** : array<auto(TT)>
- **b** : array<auto(TT)>

`algorithm::`**`combine(a: auto; b: auto) : auto`**`()`

`algorithm::`**`erase_all(arr: auto; value: auto) : auto`**`()`

Erases all elements equal to value from arr in O(n) time. Uses swap to support non-copyable types. Removed elements are swapped to the tail and properly finalized by resize.

Arguments

- **arr** : auto
- **value** : auto

fill

`algorithm::`**`fill(a: array<auto(TT)>; value: TT) : auto`**`()`

Sets all elements of the array to the given value using clone.

Arguments

- **a** : array<auto(TT)>
- **value** : TT

`algorithm::`**`fill(a: auto; value: auto) : auto`**`()`

---

is_sorted

`algorithm::is_sorted(a: auto) : bool()`

Returns true if the array is sorted in non-descending order.

Arguments

- **a** : auto

`algorithm::is_sorted(a: array<auto(TT)>) : bool()`

`algorithm::is_sorted(a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : bool()`

---

max_element

`algorithm::max_element(a: auto) : int()`

Returns the index of the maximum element in the array, or -1 if the array is empty.

Arguments

- **a** : auto

`algorithm::max_element(a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : int()`

`algorithm::max_element(a: array<auto(TT)>) : int()`

---

min_element

`algorithm::min_element(a: array<auto(TT)>; less: block<(a:TT;b:TT):bool>) : int()`

Returns the index of the minimum element according to the provided less function, or -1 if the array is empty.

Arguments

- **a** : array<auto(TT)>
- **less** : block<(a:TT;b:TT):bool>

`algorithm::min_element(a: auto) : int()`

`algorithm::min_element(a: array<auto(TT)>) : int()`

---

reverse

`algorithm::reverse(a: auto) : auto()`

Reverses the elements of array a in place.

Arguments

- **a** : auto

algorithm::**reverse(a: array<auto>) : auto**()

---

rotate

algorithm::**rotate(a: array<auto>; mid: int) : auto**()

Rotates the array so that the element at index mid becomes the first element. Elements before mid are moved to the end.

Arguments

- **a** : array<auto>
- **mid** : int

algorithm::**rotate(a: auto; mid: int) : auto**()

---

algorithm::**sort_unique(a: array<auto(TT)>) : auto**()

Returns an array of elements from a, sorted and with duplicates removed. The elements are sorted in ascending order. The resulting array has only unique elements.

Arguments

- **a** : array<auto(TT)>

algorithm::**topological_sort(nodes: array<auto(Node)>) : auto**()

Topological sort of a graph. Each node has an id, and set (table with no values) of dependencies. Dependency *before* represents a link from a node, which should appear in the sorted list before the node. Returns a sorted list of nodes.

Arguments

- **nodes** : array<auto(Node)>

algorithm::**unique(a: array<auto(TT)>) : auto**()

Returns an array with adjacent duplicate elements removed. The array should be sorted first if all duplicates need to be removed.

Arguments

- **a** : array<auto(TT)>

6.1.3 Table manipulation

- *difference (a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>*
- *identical (a: table<auto(TT), void>; b: table<auto(TT), void>) : bool*
- *intersection (a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>*
- *is_subset (a: table<auto(TT), void>; b: table<auto(TT), void>) : bool*
- *symmetric_difference (a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>*
- *union (a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>*

algorithm::**difference(a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>**()

Returns the difference of two sets.

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

algorithm::**identical(a: table<auto(TT), void>; b: table<auto(TT), void>) : bool**()

Returns true if the two sets are identical.

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

algorithm::**intersection(a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>**()

Returns the intersection of two sets.

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

algorithm::**is_subset(a: table<auto(TT), void>; b: table<auto(TT), void>) : bool**()

Returns true if all elements of a are contained in b.

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

algorithm::**symmetric_difference(a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>**()

Returns the symmetric difference of two sets (elements in either set but not both).

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

algorithm::**union(a: table<auto(TT), void>; b: table<auto(TT), void>) : table<TT, void>**()

Returns the union of two sets.

Arguments

- **a** : table<auto(TT);void>
- **b** : table<auto(TT);void>

6.2 Functional programming library

The FUNCTIONAL module implements lazy iterator adapters and higher-order function utilities including `filter`, `map`, `reduce`, `fold`, `scan`, `flatten`, `flat_map`, `enumerate`, `chain`, `pairwise`, `iterate`, `islice`, `cycle`, `repeat`, `sorted`, `sum`, `any`, `all`, `tap`, `for_each`, `find`, `find_index`, and `partition`.

See tutorial_functional for a hands-on tutorial.

All functions and symbols are in "functional" module, use require to get access to it.

```
require daslib/functional
```

Example:

```
require daslib/functional

    [export]
    def main() {
        var src <- [iterator for (x in range(6)); x]
        var evens <- filter(src, @(x : int) : bool { return x % 2 == 0; })
        for (v in evens) {
            print("{v} ")
        }
        print("\n")
    }
    // output:
    // 0 2 4
```

6.2.1 Transformations

- *filter (var src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : auto*
- *filter (var src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : auto*
- *flat_map (var src: iterator<auto(TT)>; blk: lambda<(what:TT):auto(QQ)>) : auto*
- *flat_map (var src: iterator<auto(TT)>; blk: function<(what:TT):auto(QQ)>) : auto*
- *flatten (var it: iterator<auto(TT)>) : auto*
- *map (var src: iterator<auto(TT)>; blk: function<(what:TT):auto(QQ)>) : auto*
- *map (var src: iterator<auto(TT)>; blk: lambda<(what:TT):auto(QQ)>) : auto*
- *scan (var src: iterator<auto(TT)>; seed: auto(AGG); blk: function<(acc:AGG;x:TT):AGG>) : auto*
- *scan (var src: iterator<auto(TT)>; seed: auto(AGG); blk: lambda<(acc:AGG;x:TT):AGG>) : auto*
- *sorted (var it: iterator<auto(TT)>) : auto*
- *sorted (var arr: array<auto>) : auto*

filter

`functional::filter(src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : auto()`

iterates over *src* and yields only those elements for which *blk* returns true

Arguments

- **src** : iterator<auto(TT)>
- **blk** : function<(what:TT):bool>

`functional::filter(src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : auto()`

---

flat_map

`functional::flat_map(src: iterator<auto(TT)>; blk: lambda<(what:TT):auto(QQ)>) : auto()`

maps each element to an iterator, then flattens the results one level

Arguments

- **src** : iterator<auto(TT)>
- **blk** : lambda<(what:TT):auto(QQ)>

`functional::flat_map(src: iterator<auto(TT)>; blk: function<(what:TT):auto(QQ)>) : auto()`

`functional::flatten(it: iterator<auto(TT)>) : auto()`

iterates over *it*, then iterates over each element of each element of *it* and yields it

Arguments

- **it** : iterator<auto(TT)>

map

`functional::map(src: iterator<auto(TT)>; blk: function<(what:TT):auto(QQ)>) : auto()`

iterates over *src* and yields the result of *blk* for each element

Arguments

- **src** : iterator<auto(TT)>
- **blk** : function<(what:TT):auto(QQ)>

`functional::map(src: iterator<auto(TT)>; blk: lambda<(what:TT):auto(QQ)>) : auto()`

---

scan

`functional::scan(src: iterator<auto(TT)>; seed: auto(AGG); blk: function<(acc:AGG;x:TT):AGG>) : auto()`

yields every intermediate accumulator value, starting from *seed*

Arguments

- **src** : iterator<auto(TT)>
- **seed** : auto(AGG)
- **blk** : function<(acc:AGG;x:TT):AGG>

`functional::scan(src: iterator<auto(TT)>; seed: auto(AGG); blk: lambda<(acc:AGG;x:TT):AGG>) : auto()`

---

sorted

`functional::sorted(it: iterator<auto(TT)>) : auto()`

iterates over input and returns it sorted version

Arguments

- **it** : iterator<auto(TT)>

`functional::sorted(arr: array<auto>) : auto()`

6.2.2 Aggregation

- *all (var it: iterator<auto(TT)>) : auto*
- *all (it: auto) : auto*
- *any (it: auto) : auto*
- *any (var it: iterator<auto(TT)>) : auto*
- *fold (var it: iterator<auto(TT)>; seed: auto(AGG); blk: lambda<(acc:AGG;x:TT):AGG>) : auto*
- *fold (var it: iterator<auto(TT)>; seed: auto(AGG); blk: block<(acc:AGG;x:TT):AGG>) : auto*
- *fold (var it: iterator<auto(TT)>; seed: auto(AGG); blk: function<(acc:AGG;x:TT):AGG>) : auto*
- *reduce (var it: iterator<auto(TT)>; blk: function<(left:TT;right:TT):TT>) : auto*
- *reduce (var it: iterator<auto(TT)>; blk: lambda<(left:TT;right:TT):TT>) : auto*
- *reduce (var it: iterator<auto(TT)>; blk: block<(left:TT;right:TT):TT>) : auto*
- *reduce_or_default (var it: iterator<auto(TT)>; blk: function<(left:TT;right:TT):TT>; default_value: TT) : auto*
- *reduce_or_default (var it: iterator<auto(TT)>; default_value: TT; blk: block<(left:TT;right:TT):TT>) : auto*
- *reduce_or_default (var it: iterator<auto(TT)>; blk: lambda<(left:TT;right:TT):TT>; default_value: TT) : auto*
- *sum (var it: iterator<auto(TT)>) : auto*

all

`functional::all(it: iterator<auto(TT)>) : auto()`

iterates over *it* and yields true if all elements are true

Arguments

- **it** : iterator<auto(TT)>

`functional::all(it: auto) : auto()`

---

any

`functional::any(it: auto) : auto()`

iterates over *it* and yields true if any element is true

Arguments

- **it** : auto

`functional::any(it: iterator<auto(TT)>) : auto()`

---

fold

`functional::fold(it: iterator<auto(TT)>; seed: auto(AGG); blk: lambda<(acc:AGG;x:TT):AGG>) : auto()`

combines elements left-to-right starting from *seed*

Arguments

- **it** : iterator<auto(TT)>
- **seed** : auto(AGG)
- **blk** : lambda<(acc:AGG;x:TT):AGG>

`functional::fold(it: iterator<auto(TT)>; seed: auto(AGG); blk: block<(acc:AGG;x:TT):AGG>) : auto()`

`functional::fold(it: iterator<auto(TT)>; seed: auto(AGG); blk: function<(acc:AGG;x:TT):AGG>) : auto()`

---

reduce

`functional::reduce(it: iterator<auto(TT)>; blk: function<(left:TT;right:TT):TT>) : auto()`

iterates over *it* and yields the reduced (combined) result of *blk* for each element and previous reduction result

Arguments

- **it** : iterator<auto(TT)>
- **blk** : function<(left:TT;right:TT):TT>

functional::**reduce(it: iterator<auto(TT)>; blk: lambda<(left:TT;right:TT):TT>) : auto**()

functional::**reduce(it: iterator<auto(TT)>; blk: block<(left:TT;right:TT):TT>) : auto**()

---

reduce_or_default

functional::**reduce_or_default(it: iterator<auto(TT)>; blk: function<(left:TT;right:TT):TT>; default_valu**

like reduce, but returns *default_value* on empty input

Arguments

- **it** : iterator<auto(TT)>
- **blk** : function<(left:TT;right:TT):TT>
- **default_value** : TT

functional::**reduce_or_default(it: iterator<auto(TT)>; default_value: TT; blk: block<(left:TT;right:TT):**

functional::**reduce_or_default(it: iterator<auto(TT)>; blk: lambda<(left:TT;right:TT):TT>; default_value**

---

functional::**sum(it: iterator<auto(TT)>) : auto**()

iterates over *it* and yields the sum of all elements same as reduce(it, @(a,b) => a + b)

Arguments

- **it** : iterator<auto(TT)>

6.2.3 Search and split

- *find (var src: iterator<auto(TT)>; blk: function<(what:TT):bool>; default_value: TT) : auto*
- *find (var src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>; default_value: TT) : auto*
- *find (var src: iterator<auto(TT)>; default_value: TT; blk: block<(what:TT):bool>) : auto*
- *find_index (var src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : int*
- *find_index (var src: iterator<auto(TT)>; blk: block<(what:TT):bool>) : int*
- *find_index (var src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : int*
- *partition (var src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : tuple<array<TT>;array<TT>>*
- *partition (var src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : tuple<array<TT>;array<TT>>*
- *partition (var src: iterator<auto(TT)>; blk: block<(what:TT):bool>) : tuple<array<TT>;array<TT>>*

find

`functional::find(src: iterator<auto(TT)>; blk: function<(what:TT):bool>; default_value: TT) : auto()`

returns the first element for which *blk* returns true, or *default_value*

Arguments

- **src** : iterator<auto(TT)>
- **blk** : function<(what:TT):bool>
- **default_value** : TT

`functional::find(src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>; default_value: TT) : auto()`

`functional::find(src: iterator<auto(TT)>; default_value: TT; blk: block<(what:TT):bool>) : auto()`

---

find_index

`functional::find_index(src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : int()`

returns the index of the first element for which *blk* returns true, or -1

Arguments

- **src** : iterator<auto(TT)>
- **blk** : lambda<(what:TT):bool>

`functional::find_index(src: iterator<auto(TT)>; blk: block<(what:TT):bool>) : int()`

`functional::find_index(src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : int()`

---

partition

`functional::partition(src: iterator<auto(TT)>; blk: lambda<(what:TT):bool>) : tuple<array<TT>;array<TT>`

splits elements into *(matching, non_matching)* arrays

Arguments

- **src** : iterator<auto(TT)>
- **blk** : lambda<(what:TT):bool>

`functional::partition(src: iterator<auto(TT)>; blk: function<(what:TT):bool>) : tuple<array<TT>;array<T`

`functional::partition(src: iterator<auto(TT)>; blk: block<(what:TT):bool>) : tuple<array<TT>;array<TT>>`

6.2.4 Iteration

- *echo (x: auto; extra: string = "n") : auto*
- *enumerate (var src: iterator<auto(TT)>) : auto*
- *for_each (var src: iterator<auto(TT)>; blk: function<(what:TT):void>) : auto*
- *for_each (var src: iterator<auto(TT)>; blk: lambda<(what:TT):void>) : auto*
- *for_each (var src: iterator<auto(TT)>; blk: block<(what:TT):void>) : auto*
- *tap (var src: iterator<auto(TT)>; blk: function<(what:TT):void>) : auto*
- *tap (var src: iterator<auto(TT)>; blk: lambda<(what:TT):void>) : auto*

`functional::echo(x: auto; extra: string = "\n") : auto()`

prints *x* to the output with *extra* appended, then returns *x* unchanged. Non-destructive — safe to use in expression chains.

Arguments

- **x** : auto
- **extra** : string

`functional::enumerate(src: iterator<auto(TT)>) : auto()`

yields tuples of *(index, element)* for each element in *src*

Arguments

- **src** : iterator<auto(TT)>

for_each

`functional::for_each(src: iterator<auto(TT)>; blk: function<(what:TT):void>) : auto()`

invokes *blk* on every element of *src*

Arguments

- **src** : iterator<auto(TT)>
- **blk** : function<(what:TT):void>

`functional::for_each(src: iterator<auto(TT)>; blk: lambda<(what:TT):void>) : auto()`

`functional::for_each(src: iterator<auto(TT)>; blk: block<(what:TT):void>) : auto()`

tap

`functional::tap(src: iterator<auto(TT)>; blk: function<(what:TT):void>) : auto()`

yields every element unchanged, calling *blk* on each as a side-effect

Arguments

- **src** : iterator<auto(TT)>
- **blk** : function<(what:TT):void>

`functional::tap(src: iterator<auto(TT)>; blk: lambda<(what:TT):void>) : auto()`

6.2.5 Generators

- *chain (var a: iterator<auto(TT)>; var b: iterator<auto(TT)>) : auto*
- *cycle (var src: iterator<auto(TT)>) : auto*
- *islice (var src: iterator<auto(TT)>; start: int; stop: int) : auto*
- *iterate (seed: auto(TT); var blk: function<(what:TT):TT>) : auto*
- *iterate (seed: auto(TT); var blk: lambda<(what:TT):TT>) : auto*
- *pairwise (var src: iterator<auto(TT)>) : auto*
- *repeat (value: auto(TT); var count: int = -(1)) : auto*
- *repeat_ref (value: auto(TT); var total: int) : auto*

`functional::chain(a: iterator<auto(TT)>; b: iterator<auto(TT)>) : auto()`

yields all elements of *a*, then all elements of *b*

Arguments

- **a** : iterator<auto(TT)>
- **b** : iterator<auto(TT)>

`functional::cycle(src: iterator<auto(TT)>) : auto()`

endlessly iterates over *src*

Arguments

- **src** : iterator<auto(TT)>

`functional::islice(src: iterator<auto(TT)>; start: int; stop: int) : auto()`

iterates over *src* and yields only the elements in the range [start,stop)

Arguments

- **src** : iterator<auto(TT)>
- **start** : int
- **stop** : int

iterate

`functional::iterate(seed: auto(TT); blk: function<(what:TT):TT>) : auto()`

yields *seed*, *f(seed)*, *f(f(seed))*, … infinitely.

Arguments

- **seed** : auto(TT)
- **blk** : function<(what:TT):TT>

`functional::iterate(seed: auto(TT); blk: lambda<(what:TT):TT>) : auto()`

`functional::pairwise(src: iterator<auto(TT)>) : auto()`

yields consecutive pairs: *(a,b)*, *(b,c)*, *(c,d)*, …

Arguments

- **src** : iterator<auto(TT)>

`functional::repeat(value: auto(TT); count: int = -(1)) : auto()`

yields *value count* times. If *count* is negative, repeats forever.

Arguments

- **value** : auto(TT)
- **count** : int

`functional::repeat_ref(value: auto(TT); total: int) : auto()`

yields *value* by reference *count* times

Arguments

- **value** : auto(TT)
- **total** : int

6.2.6 Predicates

- *is_equal (a: auto; b: auto) : auto*
- *is_not_equal (a: auto; b: auto) : auto*
- *not (x: auto) : auto*

`functional::is_equal(a: auto; b: auto) : auto()`

yields true if *a* and *b* are equal

Arguments

- **a** : auto
- **b** : auto

`functional::`**`is_not_equal(a: auto; b: auto) : auto`**`()`

yields true if *a* and *b* are not equal

Arguments

- **a** : auto
- **b** : auto

`functional::`**`not(x: auto) : auto`**`()`

yields !x

Arguments

- **x** : auto

6.3 LINQ

The LINQ module provides query-style operations on sequences: filtering (`where_`), projection (`select`), sorting (`order`, `order_by`), deduplication (`distinct`), pagination (`skip`, `take`), aggregation (`sum`, `average`, `aggregate`), and element access (`first`, `last`).

See also *Boost module for LINQ* for pipe-syntax macros with underscore shorthand. See tutorial_linq for a hands-on tutorial.

All functions and symbols are in "linq" module, use require to get access to it.

```
require daslib/linq
```

Example:

```
require daslib/linq

    [export]
    def main() {
        var src <- [iterator for (x in range(10)); x]
        var evens <- where_(src, $(x : int) : bool { return x % 2 == 0; })
        for (v in evens) {
            print("{v} ")
        }
        print("\n")
    }
    // output:
    // 0 2 4 6 8
```

6.3.1 Sorting data

- *order (var a: iterator<auto(TT)>) : iterator<TT>*
- *order (var a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : iterator<TT>*
- *order (a: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>*
- *order (arr: array<auto(TT)>) : array<TT>*
- *order_by (a: array<auto(TT)>; key: auto) : array<TT>*
- *order_by (var a: iterator<auto(TT)>; key: auto) : iterator<TT>*
- *order_by_descending (var a: iterator<auto(TT)>; key: auto) : iterator<TT>*
- *order_by_descending (a: array<auto(TT)>; key: auto) : array<TT>*
- *order_by_descending_inplace (var buffer: array<auto(TT)>; key: auto) : auto*
- *order_by_descending_to_array (var a: iterator<auto(TT)>; key: auto) : array<TT>*
- *order_by_inplace (var buffer: array<auto(TT)>; key: auto) : auto*
- *order_by_to_array (var a: iterator<auto(TT)>; key: auto) : array<TT>*
- *order_descending (var a: iterator<auto(TT)>) : iterator<TT>*
- *order_descending (a: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>*
- *order_descending (arr: array<auto(TT)>) : array<TT>*
- *order_descending (var a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : iterator<TT>*
- *order_descending_inplace (var buffer: array<auto(TT)>) : auto*
- *order_descending_inplace (var buffer: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : auto*
- *order_descending_to_array (var a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>*
- *order_descending_to_array (var a: iterator<auto(TT)>) : array<TT>*
- *order_inplace (var buffer: array<auto(TT)>) : auto*
- *order_inplace (var buffer: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : auto*
- *order_to_array (var a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>*
- *order_to_array (var a: iterator<auto(TT)>) : array<TT>*
- *order_unique_folded (var a: iterator<auto(TT)>) : array<TT>*
- *order_unique_folded (var a: array<auto(TT)>) : array<TT>*
- *order_unique_folded_inplace (var a: array<auto(TT)>) : auto*
- *reverse (a: array<auto(TT)>) : array<TT>*
- *reverse (var a: iterator<auto(TT)>) : iterator<TT>*
- *reverse_inplace (var buffer: array<auto(TT)>) : auto*
- *reverse_to_array (var a: iterator<auto(TT)>) : array<TT>*

order

`linq::order(a: iterator<auto(TT)>) : iterator<TT>()`

Sorts an iterator

Arguments

- **a** : iterator<auto(TT)>

`linq::order(a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : iterator<TT>()`

`linq::order(a: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>()`

`linq::order(arr: array<auto(TT)>) : array<TT>()`

---

order_by

`linq::order_by(a: array<auto(TT)>; key: auto) : array<TT>()`

Sorts an array

Arguments

- **a** : array<auto(TT)>
- **key** : auto

`linq::order_by(a: iterator<auto(TT)>; key: auto) : iterator<TT>()`

---

order_by_descending

`linq::order_by_descending(a: iterator<auto(TT)>; key: auto) : iterator<TT>()`

Sorts an iterator in descending order

Arguments

- **a** : iterator<auto(TT)>
- **key** : auto

`linq::order_by_descending(a: array<auto(TT)>; key: auto) : array<TT>()`

`linq::order_by_descending_inplace(buffer: array<auto(TT)>; key: auto) : auto()`

Sorts an array in descending order in place

Arguments

- **buffer** : array<auto(TT)>
- **key** : auto

linq::**order_by_descending_to_array(a: iterator<auto(TT)>; key: auto) : array<TT>()**

Sorts an iterator in descending order and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **key** : auto

linq::**order_by_inplace(buffer: array<auto(TT)>; key: auto) : auto()**

Sorts an array in place

Arguments

- **buffer** : array<auto(TT)>
- **key** : auto

linq::**order_by_to_array(a: iterator<auto(TT)>; key: auto) : array<TT>()**

Sorts an iterator and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **key** : auto

order_descending

linq::**order_descending(a: iterator<auto(TT)>) : iterator<TT>()**

Sorts an iterator in descending order

Arguments

- **a** : iterator<auto(TT)>

linq::**order_descending(a: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>()**

linq::**order_descending(arr: array<auto(TT)>) : array<TT>()**

linq::**order_descending(a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : iterator<TT>()**

---

order_descending_inplace

linq::**order_descending_inplace(buffer: array<auto(TT)>) : auto()**

Sorts an array in descending order in place

Arguments

- **buffer** : array<auto(TT)>

linq::**order_descending_inplace(buffer: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : auto()**

---

order_descending_to_array

linq::**order_descending_to_array(a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>()**

Sorts an iterator in descending order and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **fun** : block<(v1:TT;v2:TT):bool>

linq::**order_descending_to_array(a: iterator<auto(TT)>) : array<TT>()**

---

order_inplace

linq::**order_inplace(buffer: array<auto(TT)>) : auto()**

Sorts an array in place

Arguments

- **buffer** : array<auto(TT)>

linq::**order_inplace(buffer: array<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : auto()**

---

order_to_array

linq::**order_to_array(a: iterator<auto(TT)>; fun: block<(v1:TT;v2:TT):bool>) : array<TT>()**

Sorts an iterator and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **fun** : block<(v1:TT;v2:TT):bool>

linq::**order_to_array(a: iterator<auto(TT)>) : array<TT>()**

---

order_unique_folded

linq::**order_unique_folded(a: iterator<auto(TT)>) : array<TT>()**

sort and remove duplicate elements from an iterator

Arguments

- **a** : iterator<auto(TT)>

linq::**order_unique_folded(a: array<auto(TT)>) : array<TT>()**

`linq::order_unique_folded_inplace(a: array<auto(TT)>) : auto()`

sort and remove duplicate elements from an array

Arguments

- **a** : array<auto(TT)>

reverse

`linq::reverse(a: array<auto(TT)>) : array<TT>()`

Reverses an array

Arguments

- **a** : array<auto(TT)>

`linq::reverse(a: iterator<auto(TT)>) : iterator<TT>()`

`linq::reverse_inplace(buffer: array<auto(TT)>) : auto()`

Reverses an array in place

Arguments

- **buffer** : array<auto(TT)>

`linq::reverse_to_array(a: iterator<auto(TT)>) : array<TT>()`

Reverses an iterator and returns an array

Arguments

- **a** : iterator<auto(TT)>

6.3.2 Set operations

- *distinct (a: array<auto(TT)>) : array<TT>*
- *distinct (var a: iterator<auto(TT)>) : iterator<TT>*
- *distinct_by (a: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *distinct_by (var a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>*
- *distinct_by_inplace (var a: array<auto(TT)>; key: block<(arg:TT):auto>) : auto*
- *distinct_by_to_array (var a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *distinct_inplace (var a: array<auto(TT)>) : auto*
- *distinct_to_array (var a: iterator<auto(TT)>) : array<TT>*
- *except (var src: iterator<auto(TT)>; var exclude: iterator<auto(TT)>) : iterator<TT>*
- *except (src: array<auto(TT)>; exclude: array<auto(TT)>) : array<TT>*
- *except_by (src: array<auto(TT)>; exclude: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *except_by (var src: iterator<auto(TT)>; var exclude: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>*

- *except_by_to_array (var src: iterator<auto(TT)>; var exclude: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *except_to_array (var src: iterator<auto(TT)>; var exclude: iterator<auto(TT)>) : array<TT>*
- *intersect (srca: array<auto(TT)>; srcb: array<auto(TT)>) : array<TT>*
- *intersect (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>) : iterator<TT>*
- *intersect_by (srca: array<auto(TT)>; srcb: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *intersect_by (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>*
- *intersect_by_to_array (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *intersect_to_array (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>) : array<TT>*
- *union (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>) : iterator<TT>*
- *union (var srca: array<auto(TT)>; var srcb: array<auto(TT)>) : array<TT>*
- *union_by (srca: array<auto(TT)>; srcb: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *union_by (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>*
- *union_by_to_array (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *union_to_array (var srca: iterator<auto(TT)>; var srcb: iterator<auto(TT)>) : array<TT>*
- *unique (a: array<auto(TT)>) : array<TT>*
- *unique (a: iterator<auto(TT)>) : iterator<TT>*
- *unique_by (a: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *unique_by (a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>*
- *unique_by_inplace (var a: array<auto(TT)>; key: block<(arg:TT):auto>) : auto*
- *unique_by_to_array (a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>*
- *unique_inplace (var a: array<auto(TT)>) : auto*
- *unique_key (a: auto) : auto*
- *unique_to_array (a: iterator<auto(TT)>) : array<TT>*

distinct

`linq::`**`distinct(a: array<auto(TT)>) : array<TT>`**`()`

Returns distinct elements from an array

Arguments

- **a** : array<auto(TT)>

`linq::`**`distinct(a: iterator<auto(TT)>) : iterator<TT>`**`()`

distinct_by

`linq::distinct_by(a: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>()`

Returns distinct elements from an array based on a key

Arguments

- **a** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::distinct_by(a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>()`

`linq::distinct_by_inplace(a: array<auto(TT)>; key: block<(arg:TT):auto>) : auto()`

Returns distinct elements from an array based on a key in place

Arguments

- **a** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::distinct_by_to_array(a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>()`

Returns distinct elements from an iterator based on a key and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::distinct_inplace(a: array<auto(TT)>) : auto()`

Returns distinct elements from an array in place

Arguments

- **a** : array<auto(TT)>

`linq::distinct_to_array(a: iterator<auto(TT)>) : array<TT>()`

Returns distinct elements from an iterator and returns an array

Arguments

- **a** : iterator<auto(TT)>

except

`linq::except(src: iterator<auto(TT)>; exclude: iterator<auto(TT)>) : iterator<TT>()`

Returns elements from the first iterator that are not in the second iterator

Arguments

- **src** : iterator<auto(TT)>
- **exclude** : iterator<auto(TT)>

`linq::except(src: array<auto(TT)>; exclude: array<auto(TT)>) : array<TT>()`

---

except_by

`linq::except_by(src: array<auto(TT)>; exclude: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>(`

Returns elements from the first array that are not in the second array by key

Arguments

- **src** : array<auto(TT)>
- **exclude** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::except_by(src: iterator<auto(TT)>; exclude: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iter`

`linq::except_by_to_array(src: iterator<auto(TT)>; exclude: iterator<auto(TT)>; key: block<(arg:TT):auto`

Returns elements from the first iterator that are not in the second iterator by key and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **exclude** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::except_to_array(src: iterator<auto(TT)>; exclude: iterator<auto(TT)>) : array<TT>()`

Returns elements from the first iterator that are not in the second iterator and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **exclude** : iterator<auto(TT)>

intersect

`linq::intersect(srca: array<auto(TT)>; srcb: array<auto(TT)>) : array<TT>()`

Returns elements that are present in both arrays

Arguments

- **srca** : array<auto(TT)>
- **srcb** : array<auto(TT)>

`linq::intersect(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>) : iterator<TT>()`

---

intersect_by

linq::**intersect_by(srca: array<auto(TT)>; srcb: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>**

Returns elements that are present in both arrays by key

Arguments

- **srca** : array<auto(TT)>
- **srcb** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

linq::**intersect_by(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : ite**

---

linq::**intersect_by_to_array(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>; key: block<(arg:TT):aut**

Returns elements that are present in both iterators by key and returns an array

Arguments

- **srca** : iterator<auto(TT)>
- **srcb** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

linq::**intersect_to_array(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>) : array<TT>**()

Returns elements that are present in both iterators and returns an array

Arguments

- **srca** : iterator<auto(TT)>
- **srcb** : iterator<auto(TT)>

union

linq::**union(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>) : iterator<TT>**()

Returns distinct elements from the concatenation of two iterators

Arguments

- **srca** : iterator<auto(TT)>
- **srcb** : iterator<auto(TT)>

linq::**union(srca: array<auto(TT)>; srcb: array<auto(TT)>) : array<TT>**()

---

union_by

linq::**union_by(srca: array<auto(TT)>; srcb: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>**()

Returns distinct elements from the concatenation of two arrays by key

Arguments

- **srca** : array<auto(TT)>
- **srcb** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

linq::**union_by(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterato**

---

linq::**union_by_to_array(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>; key: block<(arg:TT):auto>)**

Returns distinct elements from the concatenation of two iterators by key and returns an array

Arguments

- **srca** : iterator<auto(TT)>
- **srcb** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

linq::**union_to_array(srca: iterator<auto(TT)>; srcb: iterator<auto(TT)>) : array<TT>**()

Returns distinct elements from the concatenation of two iterators and returns an array

Arguments

- **srca** : iterator<auto(TT)>
- **srcb** : iterator<auto(TT)>

unique

linq::**unique(a: array<auto(TT)>) : array<TT>**()

sort and remove duplicate elements from an array

Arguments

- **a** : array<auto(TT)>

linq::**unique(a: iterator<auto(TT)>) : iterator<TT>**()

---

unique_by

`linq::unique_by(a: array<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>()`

sort and remove duplicate elements from an array based on a key

Arguments

- **a** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::unique_by(a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : iterator<TT>()`

`linq::unique_by_inplace(a: array<auto(TT)>; key: block<(arg:TT):auto>) : auto()`

remove duplicate elements from an array based on a key in place

Arguments

- **a** : array<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::unique_by_to_array(a: iterator<auto(TT)>; key: block<(arg:TT):auto>) : array<TT>()`

sort and remove duplicate elements from an iterator based on a key and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::unique_inplace(a: array<auto(TT)>) : auto()`

remove duplicate elements from sorted array in place

Arguments

- **a** : array<auto(TT)>

`linq::unique_key(a: auto) : auto()`

generates unique key of workhorse type for the value

Arguments

- **a** : auto

`linq::unique_to_array(a: iterator<auto(TT)>) : array<TT>()`

sort and remove duplicate elements from an iterator and returns an array

Arguments

- **a** : iterator<auto(TT)>

6.3.3 Concatenation operations

- *append (arr: array<auto(TT)>; value: TT) : array<TT>*
- *append (var it: iterator<auto(TT)>; value: TT) : iterator<TT>*
- *append_inplace (var arr: array<auto(TT)>; value: TT) : auto*
- *append_to_array (var it: iterator<auto(TT)>; value: TT) : array<TT>*
- *concat (var a: iterator<auto(TT)>; var b: iterator<auto(TT)>) : iterator<TT>*
- *concat (a: array<auto(TT)>; b: array<auto(TT)>) : array<TT>*
- *concat_inplace (var a: array<auto(TT)>; b: array<auto(TT)>) : auto*
- *concat_to_array (var a: iterator<auto(TT)>; var b: iterator<auto(TT)>) : array<TT>*
- *prepend (var it: iterator<auto(TT)>; value: TT) : iterator<TT>*
- *prepend (arr: array<auto(TT)>; value: TT) : array<TT>*
- *prepend_inplace (var arr: array<auto(TT)>; value: TT) : auto*
- *prepend_to_array (var it: iterator<auto(TT)>; value: TT) : array<TT>*

append

`linq::`**`append(arr: array<auto(TT)>; value: TT) : array<TT>`**`()`

Appends a value to the end of an array

Arguments

- **arr** : array<auto(TT)>
- **value** : TT

`linq::`**`append(it: iterator<auto(TT)>; value: TT) : iterator<TT>`**`()`

`linq::`**`append_inplace(arr: array<auto(TT)>; value: TT) : auto`**`()`

Appends a value to the end of an array in place

Arguments

- **arr** : array<auto(TT)>
- **value** : TT

`linq::`**`append_to_array(it: iterator<auto(TT)>; value: TT) : array<TT>`**`()`

Appends a value to the end of an iterator and returns an array

Arguments

- **it** : iterator<auto(TT)>
- **value** : TT

concat

`linq::concat(a: iterator<auto(TT)>; b: iterator<auto(TT)>) : iterator<TT>()`

Concatenates two iterators

Arguments

- **a** : iterator<auto(TT)>
- **b** : iterator<auto(TT)>

`linq::concat(a: array<auto(TT)>; b: array<auto(TT)>) : array<TT>()`

`linq::concat_inplace(a: array<auto(TT)>; b: array<auto(TT)>) : auto()`

Concatenates two arrays in place

Arguments

- **a** : array<auto(TT)>
- **b** : array<auto(TT)>

`linq::concat_to_array(a: iterator<auto(TT)>; b: iterator<auto(TT)>) : array<TT>()`

Concatenates two iterators and returns an array

Arguments

- **a** : iterator<auto(TT)>
- **b** : iterator<auto(TT)>

prepend

`linq::prepend(it: iterator<auto(TT)>; value: TT) : iterator<TT>()`

Prepends a value to the beginning of an iterator

Arguments

- **it** : iterator<auto(TT)>
- **value** : TT

`linq::prepend(arr: array<auto(TT)>; value: TT) : array<TT>()`

`linq::prepend_inplace(arr: array<auto(TT)>; value: TT) : auto()`

Prepends a value to the beginning of an array in place

Arguments

- **arr** : array<auto(TT)>
- **value** : TT

`linq::prepend_to_array(it: iterator<auto(TT)>; value: TT) : array<TT>()`

Prepends a value to the beginning of an iterator and returns an array

Arguments

- **it** : iterator<auto(TT)>
- **value** : TT

6.3.4 Generation operations

- *default_empty (var src: iterator<auto(TT)>) : iterator<TT>*
- *empty (var typ: auto(TT)) : iterator<TT>*
- *range_sequence (start: int; count: int) : iterator<int>*
- *repeat (element: auto(TT); count: int) : iterator<TT>*

`linq::default_empty(src: iterator<auto(TT)>) : iterator<TT>()`

Returns the elements of the iterator, or a default value if the iterator is empty

Arguments

- **src** : iterator<auto(TT)>

`linq::empty(typ: auto(TT)) : iterator<TT>()`

Returns an empty iterator of the specified type

Arguments

- **typ** : auto(TT)

`linq::range_sequence(start: int; count: int) : iterator<int>()`

Generates a sequence of integers within a specified range

Arguments

- **start** : int
- **count** : int

`linq::repeat(element: auto(TT); count: int) : iterator<TT>()`

Generates a sequence that contains one repeated value

Arguments

- **element** : auto(TT)
- **count** : int

6.3.5 Aggregation operations

- *aggregate (var src: iterator<auto(TT)>; seed: auto(AGG); func: block<(acc:AGG;x:TT):AGG>) : AGG*
- *aggregate (src: array<auto(TT)>; seed: auto(AGG); func: block<(acc:AGG;x:TT):AGG>) : AGG*
- *average (src: array<auto(TT)>) : TT*
- *average (var src: iterator<auto(TT)>) : TT*
- *count (a: array<auto(TT)>; predicate: block<(arg:TT):bool>) : int*
- *count (a: array<auto(TT)>) : int*
- *count (var a: iterator<auto(TT)>) : int*
- *count (var a: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : int*
- *long_count (a: array<auto(TT)>) : int64*
- *long_count (var a: iterator<auto(TT)>) : int64*
- *max (var src: iterator<auto(TT)>) : TT*
- *max (src: array<auto(TT)>) : TT*
- *max_by (var src: iterator<auto(TT)>; key: auto) : TT*
- *max_by (src: array<auto(TT)>; key: auto) : TT*
- *min (var src: iterator<auto(TT)>) : TT*
- *min (src: array<auto(TT)>) : TT*
- *min_by (src: array<auto(TT)>; key: auto) : TT*
- *min_by (var src: iterator<auto(TT)>; key: auto) : TT*
- *min_max (var src: iterator<auto(TT)>) : tuple<TT;TT>*
- *min_max (src: array<auto(TT)>) : tuple<TT;TT>*
- *min_max_average (src: array<auto(TT)>) : tuple<TT;TT;TT>*
- *min_max_average (var src: iterator<auto(TT)>) : tuple<TT;TT;TT>*
- *min_max_average_by (src: array<auto(TT)>; key: auto) : tuple<TT;TT;TT>*
- *min_max_average_by (var src: iterator<auto(TT)>; key: auto) : tuple<TT;TT;TT>*
- *min_max_by (var src: iterator<auto(TT)>; key: auto) : tuple<TT;TT>*
- *min_max_by (src: array<auto(TT)>; key: auto) : tuple<TT;TT>*
- *sum (src: array<auto(TT)>) : TT*
- *sum (var src: iterator<auto(TT)>) : TT*

aggregate

linq::**aggregate(src: iterator<auto(TT)>; seed: auto(AGG); func: block<(acc:AGG;x:TT):AGG>) : AGG()**

Aggregates elements in an iterator using a seed and a function

Arguments

- **src** : iterator<auto(TT)>
- **seed** : auto(AGG)
- **func** : block<(acc:AGG;x:TT):AGG>

linq::**aggregate(src: array<auto(TT)>; seed: auto(AGG); func: block<(acc:AGG;x:TT):AGG>) : AGG()**

---

average

linq::**average(src: array<auto(TT)>) : TT()**

Averages elements in an array

Arguments

- **src** : array<auto(TT)>

linq::**average(src: iterator<auto(TT)>) : TT()**

---

count

linq::**count(a: array<auto(TT)>; predicate: block<(arg:TT):bool>) : int()**

Counts elements in an array that satisfy a predicate

Arguments

- **a** : array<auto(TT)>
- **predicate** : block<(arg:TT):bool>

linq::**count(a: array<auto(TT)>) : int()**

linq::**count(a: iterator<auto(TT)>) : int()**

linq::**count(a: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : int()**

---

long_count

`linq::long_count(a: array<auto(TT)>) : int64()`

Counts elements in an array, using a long integer

Arguments

- **a** : array<auto(TT)>

`linq::long_count(a: iterator<auto(TT)>) : int64()`

---

max

`linq::max(src: iterator<auto(TT)>) : TT()`

Finds the maximum element in an iterator

Arguments

- **src** : iterator<auto(TT)>

`linq::max(src: array<auto(TT)>) : TT()`

---

max_by

`linq::max_by(src: iterator<auto(TT)>; key: auto) : TT()`

Finds the maximum element in an iterator by key

Arguments

- **src** : iterator<auto(TT)>
- **key** : auto

`linq::max_by(src: array<auto(TT)>; key: auto) : TT()`

---

min

`linq::min(src: iterator<auto(TT)>) : TT()`

Finds the minimum element in an iterator

Arguments

- **src** : iterator<auto(TT)>

`linq::min(src: array<auto(TT)>) : TT()`

---

min_by

linq::**min_by(src: array<auto(TT)>; key: auto) : TT()**

Finds the minimum element in an array by key

Arguments

- **src** : array<auto(TT)>
- **key** : auto

linq::**min_by(src: iterator<auto(TT)>; key: auto) : TT()**

min_max

linq::**min_max(src: iterator<auto(TT)>) : tuple<TT;TT>()**

Finds the minimum and maximum elements in an iterator

Arguments

- **src** : iterator<auto(TT)>

linq::**min_max(src: array<auto(TT)>) : tuple<TT;TT>()**

min_max_average

linq::**min_max_average(src: array<auto(TT)>) : tuple<TT;TT;TT>()**

Finds the minimum, maximum, and average elements in an array

Arguments

- **src** : array<auto(TT)>

linq::**min_max_average(src: iterator<auto(TT)>) : tuple<TT;TT;TT>()**

min_max_average_by

linq::**min_max_average_by(src: array<auto(TT)>; key: auto) : tuple<TT;TT;TT>()**

Finds the minimum, maximum, and average elements in an array by key

Arguments

- **src** : array<auto(TT)>
- **key** : auto

linq::**min_max_average_by(src: iterator<auto(TT)>; key: auto) : tuple<TT;TT;TT>()**

min_max_by

linq::**min_max_by(src: iterator<auto(TT)>; key: auto) : tuple<TT;TT>**()

Finds the minimum and maximum elements in an iterator by key

Arguments

- **src** : iterator<auto(TT)>
- **key** : auto

linq::**min_max_by(src: array<auto(TT)>; key: auto) : tuple<TT;TT>**()

---

sum

linq::**sum(src: array<auto(TT)>) : TT**()

Sums elements in an array

Arguments

- **src** : array<auto(TT)>

linq::**sum(src: iterator<auto(TT)>) : TT**()

6.3.6 Filtering data

- *where_ (src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*
- *where_ (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>*
- *where_to_array (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*

where

linq::**where_(src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>**()

Filters elements in an array based on a predicate

Arguments

- **src** : array<auto(TT)>
- **predicate** : block<(arg:TT):bool>

linq::**where_(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>**()

---

linq::**where_to_array(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>**()

Filters elements in an iterator based on a predicate and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **predicate** : block<(arg:TT):bool>

6.3.7 Partitioning data

- *chunk (src: array<auto(TT)>; size: int) : array<array<TT>>*
- *chunk (var src: iterator<auto(TT)>; size: int) : iterator<array<TT>>*
- *chunk_to_array (var src: iterator<auto(TT)>; size: int) : array<array<TT>>*
- *skip (var src: iterator<auto(TT)>; var total: int) : iterator<TT>*
- *skip (arr: array<auto(TT)>; var total: int) : array<TT>*
- *skip_inplace (var arr: array<auto(TT)>; var total: int) : auto*
- *skip_last (var src: iterator<auto(TT)>; var total: int) : iterator<TT>*
- *skip_last (arr: array<auto(TT)>; var total: int) : array<TT>*
- *skip_last_inplace (var arr: array<auto(TT)>; var total: int) : auto*
- *skip_last_to_array (var src: iterator<auto(TT)>; var total: int) : array<TT>*
- *skip_to_array (var src: iterator<auto(TT)>; var total: int) : array<TT>*
- *skip_while (src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*
- *skip_while (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>*
- *skip_while_to_array (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*
- *take (src: array<auto(TT)>; from: range) : array<TT>*
- *take (var src: iterator<auto(TT)>; from: range) : iterator<TT>*
- *take (var src: iterator<auto(TT)>; var total: int) : iterator<TT>*
- *take (arr: array<auto(TT)>; var total: int) : array<TT>*
- *take_inplace (var arr: array<auto(TT)>; from: range) : auto*
- *take_inplace (var arr: array<auto(TT)>; var total: int) : auto*
- *take_last (var src: iterator<auto(TT)>; var total: int) : iterator<TT>*
- *take_last (arr: array<auto(TT)>; var total: int) : array<TT>*
- *take_last_inplace (var arr: array<auto(TT)>; var total: int) : auto*
- *take_last_to_array (var src: iterator<auto(TT)>; var total: int) : array<TT>*
- *take_to_array (var src: iterator<auto(TT)>; var total: int) : array<TT>*
- *take_to_array (var src: iterator<auto(TT)>; from: range) : array<TT>*
- *take_while (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>*
- *take_while (src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*
- *take_while_to_array (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>*

chunk

`linq::chunk(src: array<auto(TT)>; size: int) : array<array<TT>>()`

Splits an array into chunks of a specified size

Arguments

- **src** : array<auto(TT)>
- **size** : int

`linq::chunk(src: iterator<auto(TT)>; size: int) : iterator<array<TT>>()`

`linq::chunk_to_array(src: iterator<auto(TT)>; size: int) : array<array<TT>>()`

Splits an iterator into chunks of a specified size and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **size** : int

skip

`linq::skip(src: iterator<auto(TT)>; total: int) : iterator<TT>()`

Yields all but the first *total* elements

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

`linq::skip(arr: array<auto(TT)>; total: int) : array<TT>()`

`linq::skip_inplace(arr: array<auto(TT)>; total: int) : auto()`

Removes the first *total* elements from an array in place

Arguments

- **arr** : array<auto(TT)>
- **total** : int

skip_last

`linq::skip_last(src: iterator<auto(TT)>; total: int) : iterator<TT>()`

Yields all but the last *total* elements from an iterator

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

`linq::skip_last(arr: array<auto(TT)>; total: int) : array<TT>()`

`linq::skip_last_inplace(arr: array<auto(TT)>; total: int) : auto()`

Removes the last *total* elements from an array in place

Arguments

- **arr** : array<auto(TT)>
- **total** : int

`linq::skip_last_to_array(src: iterator<auto(TT)>; total: int) : array<TT>()`

Yields all but the last *total* elements from an iterator and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

`linq::skip_to_array(src: iterator<auto(TT)>; total: int) : array<TT>()`

Yields all but the first *total* elements and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

skip_while

`linq::skip_while(src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>()`

Skips all elements of an array while the predicate is true

Arguments

- **src** : array<auto(TT)>
- **predicate** : block<(arg:TT):bool>

`linq::skip_while(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>()`

`linq::skip_while_to_array(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>()`

Skips all elements of an iterator while the predicate is true and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **predicate** : block<(arg:TT):bool>

take

`linq::take(src: array<auto(TT)>; from: range) : array<TT>()`

Yields a range of elements from an array

Arguments

- **src** : array<auto(TT)>
- **from** : range

`linq::take(src: iterator<auto(TT)>; from: range) : iterator<TT>()`

`linq::take(src: iterator<auto(TT)>; total: int) : iterator<TT>()`

`linq::take(arr: array<auto(TT)>; total: int) : array<TT>()`

---

take_inplace

`linq::take_inplace(arr: array<auto(TT)>; from: range) : auto()`

Keeps only a range of elements in an array in place

Arguments

- **arr** : array<auto(TT)>
- **from** : range

`linq::take_inplace(arr: array<auto(TT)>; total: int) : auto()`

---

take_last

`linq::take_last(src: iterator<auto(TT)>; total: int) : iterator<TT>()`

Yields only the last *total* elements from an iterator

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

`linq::take_last(arr: array<auto(TT)>; total: int) : array<TT>()`

`linq::take_last_inplace(arr: array<auto(TT)>; total: int) : auto()`

Keeps only the last *total* elements in an array in place

Arguments

- **arr** : array<auto(TT)>
- **total** : int

linq::**take_last_to_array(src: iterator<auto(TT)>; total: int) : array<TT>()**

Yields only the last *total* elements from an iterator and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

take_to_array

linq::**take_to_array(src: iterator<auto(TT)>; total: int) : array<TT>()**

Yields only the first *total* elements and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **total** : int

linq::**take_to_array(src: iterator<auto(TT)>; from: range) : array<TT>()**

---

take_while

linq::**take_while(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : iterator<TT>()**

Yields only the elements of an iterator while the predicate is true

Arguments

- **src** : iterator<auto(TT)>
- **predicate** : block<(arg:TT):bool>

linq::**take_while(src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>()**

---

linq::**take_while_to_array(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : array<TT>()**

Yields only the elements of an iterator while the predicate is true and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **predicate** : block<(arg:TT):bool>

6.3.8 Join and group operations

- *group_by (var source: iterator<auto(TT)>; key: auto; element_selector: auto; result_selector: auto) : auto*
- *group_by (source: array<auto(TT)>; key: auto; element_selector: auto; result_selector: auto) : auto*
- *group_by_to_array (var source: iterator<auto(TT)>; key: auto; element_selector: auto; result_selector: auto) : auto*
- *group_join (var srca: iterator<auto(TA)>; var srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: auto) : iterator<typedecl(result(type<TA>,type<array<TB -const -&>>))>*
- *group_join (srca: array<auto(TA)>; srcb: array<auto(TB)>; keya: auto; keyb: auto; result: auto) : array<typedecl(result(type<TA>,type<array<TB -const -&>>))>*
- *group_join_to_array (var srca: iterator<auto(TA)>; var srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: auto) : array<typedecl(result(type<TA>,type<array<TB -const -&>>))>*
- *join (srca: array<auto(TA)>; srcb: array<auto(TB)>; keya: auto; keyb: auto; result: auto) : array<typedecl(result(type<TA>,type<TB>))>*
- *join (var srca: iterator<auto(TA)>; var srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: auto) : iterator<typedecl(result(type<TA>,type<TB>))>*
- *join_to_array (var srca: iterator<auto(TA)>; var srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: auto) : array<typedecl(result(type<TA>,type<TB>))>*

group_by

`linq::group_by(source: iterator<auto(TT)>; key: auto; element_selector: auto; result_selector: auto) : a`

Groups the elements of an iterator according to a specified key selector function

Arguments

- **source** : iterator<auto(TT)>
- **key** : auto
- **element_selector** : auto
- **result_selector** : auto

`linq::group_by(source: array<auto(TT)>; key: auto; element_selector: auto; result_selector: auto) : auto`

`linq::group_by_to_array(source: iterator<auto(TT)>; key: auto; element_selector: auto; result_selector:`

Groups the elements of an iterator according to a specified key selector function and returns an array

Arguments

- **source** : iterator<auto(TT)>
- **key** : auto
- **element_selector** : auto
- **result_selector** : auto

group_join

linq::**group_join(srca: iterator<auto(TA)>; srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: au**

we pass TA, and sequence of TB to 'result'

Arguments

- **srca** : iterator<auto(TA)>
- **srcb** : iterator<auto(TB)>
- **keya** : auto
- **keyb** : auto
- **result** : auto

linq::**group_join(srca: array<auto(TA)>; srcb: array<auto(TB)>; keya: auto; keyb: auto; result: auto) :**

---

linq::**group_join_to_array(srca: iterator<auto(TA)>; srcb: iterator<auto(TB)>; keya: auto; keyb: auto; r**

we pass TA, and sequence of TB to 'result'

Arguments

- **srca** : iterator<auto(TA)>
- **srcb** : iterator<auto(TB)>
- **keya** : auto
- **keyb** : auto
- **result** : auto

join

linq::**join(srca: array<auto(TA)>; srcb: array<auto(TB)>; keya: auto; keyb: auto; result: auto) : array<**

Joins two arrays based on matching keys (inner join)

Arguments

- **srca** : array<auto(TA)>
- **srcb** : array<auto(TB)>
- **keya** : auto
- **keyb** : auto
- **result** : auto

linq::**join(srca: iterator<auto(TA)>; srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result: auto) :**

---

linq::**join_to_array(srca: iterator<auto(TA)>; srcb: iterator<auto(TB)>; keya: auto; keyb: auto; result:**

Joins two iterators based on matching keys (inner join) and returns an array

Arguments

- **srca** : iterator<auto(TA)>
- **srcb** : iterator<auto(TB)>
- **keya** : auto
- **keyb** : auto
- **result** : auto

6.3.9 Querying data

- *all (src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : bool*
- *all (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : bool*
- *any (src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : bool*
- *any (var src: iterator<auto(TT)>) : bool*
- *any (src: array<auto(TT)>) : bool*
- *any (var src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : bool*
- *contains (var src: iterator<auto(TT)>; element: TT) : bool*
- *contains (src: array<auto(TT)>; element: TT) : bool*

all

linq::**all(src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : bool**()

Returns true if all elements in the array satisfy the predicate

Arguments

- **src** : array<auto(TT)>
- **predicate** : block<(arg:TT):bool>

linq::**all(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : bool**()

---

any

linq::**any(src: array<auto(TT)>; predicate: block<(arg:TT):bool>) : bool**()

Returns true if any element in the array satisfies the predicate

Arguments

- **src** : array<auto(TT)>
- **predicate** : block<(arg:TT):bool>

`linq::any(src: iterator<auto(TT)>) : bool()`

`linq::any(src: array<auto(TT)>) : bool()`

`linq::any(src: iterator<auto(TT)>; predicate: block<(arg:TT):bool>) : bool()`

---

contains

`linq::contains(src: iterator<auto(TT)>; element: TT) : bool()`

Returns true if the element is present in the iterator

Arguments

- **src** : iterator<auto(TT)>
- **element** : TT

`linq::contains(src: array<auto(TT)>; element: TT) : bool()`

6.3.10 Element operations

- *element_at (var src: iterator<auto(TT)>; index: int) : TT*
- *element_at (src: array<auto(TT)>; index: int) : TT*
- *element_at_or_default (var src: iterator<auto(TT)>; index: int) : TT*
- *element_at_or_default (src: array<auto(TT)>; index: int) : TT*
- *first (var src: iterator<auto(TT)>) : TT*
- *first (src: array<auto(TT)>) : TT*
- *first_or_default (src: array<auto(TT)>; defaultValue: TT) : TT*
- *first_or_default (var src: iterator<auto(TT)>; defaultValue: TT) : TT*
- *last (var src: iterator<auto(TT)>) : TT*
- *last (src: array<auto(TT)>) : TT*
- *last_or_default (src: array<auto(TT)>; defaultValue: TT) : TT*
- *last_or_default (var src: iterator<auto(TT)>; defaultValue: TT) : TT*
- *single (src: array<auto(TT)>) : TT*
- *single (var src: iterator<auto(TT)>) : TT*
- *single_or_default (var src: iterator<auto(TT)>; defaultValue: TT) : TT*
- *single_or_default (src: array<auto(TT)>; defaultValue: TT) : TT*

element_at

linq::**element_at(src: iterator<auto(TT)>; index: int) : TT()**

Returns the element at the specified index

Arguments

- **src** : iterator<auto(TT)>
- **index** : int

linq::**element_at(src: array<auto(TT)>; index: int) : TT()**

---

element_at_or_default

linq::**element_at_or_default(src: iterator<auto(TT)>; index: int) : TT()**

Returns the element at the specified index, or a default value if the index is out of range

Arguments

- **src** : iterator<auto(TT)>
- **index** : int

linq::**element_at_or_default(src: array<auto(TT)>; index: int) : TT()**

---

first

linq::**first(src: iterator<auto(TT)>) : TT()**

Returns the first element of an iterator

Arguments

- **src** : iterator<auto(TT)>

linq::**first(src: array<auto(TT)>) : TT()**

---

first_or_default

linq::**first_or_default(src: array<auto(TT)>; defaultValue: TT) : TT()**

Returns the first element of an array, or a default value if the array is empty

Arguments

- **src** : array<auto(TT)>
- **defaultValue** : TT

linq::**first_or_default(src: iterator<auto(TT)>; defaultValue: TT) : TT()**

---

last

linq::**last(src: iterator<auto(TT)>) : TT()**

Returns the last element of an iterator

Arguments

- **src** : iterator<auto(TT)>

linq::**last(src: array<auto(TT)>) : TT()**

---

last_or_default

linq::**last_or_default(src: array<auto(TT)>; defaultValue: TT) : TT()**

Returns the last element of an array, or a default value if the array is empty

Arguments

- **src** : array<auto(TT)>
- **defaultValue** : TT

linq::**last_or_default(src: iterator<auto(TT)>; defaultValue: TT) : TT()**

---

single

linq::**single(src: array<auto(TT)>) : TT()**

Returns the only element of an array, and throws if there is not exactly one element

Arguments

- **src** : array<auto(TT)>

linq::**single(src: iterator<auto(TT)>) : TT()**

---

single_or_default

linq::**single_or_default(src: iterator<auto(TT)>; defaultValue: TT) : TT()**

Returns the only element of an iterator, or a default value if there is not exactly one element

Arguments

- **src** : iterator<auto(TT)>
- **defaultValue** : TT

linq::**single_or_default(src: array<auto(TT)>; defaultValue: TT) : TT()**

6.3.11 Transform operations

- *select (src: array<auto(TT)>) : array<tuple<int;TT>>*
- *select (var src: iterator<auto(TT)>) : iterator<tuple<int;TT>>*
- *select (var src: iterator<auto(TT)>; result_selector: auto) : iterator<typedecl(result_selector(type<TT>))>*
- *select (src: array<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(type<TT>))>*
- *select_many (var src: iterator<auto(TT)>; result_selector: auto) : iterator<typedecl(result_selector(iter_type(type<TT>)))>*
- *select_many (src: array<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(iter_type(type<TT>)))>*
- *select_many (src: array<auto(TT)>; collection_selector: auto; result_selector: auto) : array<typedecl(result_selector(iter_type(collection_selector(type<TT>))))>*
- *select_many (var src: iterator<auto(TT)>; collection_selector: auto; result_selector: auto) : iterator<typedecl(result_selector(iter_type(collection_selector(type<TT>))))>*
- *select_many_to_array (var src: iterator<auto(TT)>; collection_selector: auto; result_selector: auto) : array<typedecl(result_selector(iter_type(collection_selector(type<TT>))))>*
- *select_many_to_array (var src: iterator<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(iter_type(type<TT>)))>*
- *select_to_array (var src: iterator<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(type<TT>))>*
- *select_to_array (var src: iterator<auto(TT)>) : array<tuple<int;TT>>*
- *zip (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>) : iterator<tuple<TT;UU>>*
- *zip (a: array<auto(TT)>; b: array<auto(UU)>) : array<tuple<TT;UU>>*
- *zip (a: array<auto(TT)>; b: array<auto(UU)>; result_selector: block<(l:TT;r:UU):auto>) : array<typedecl(result_selector(type<TT>,type<UU>))>*
- *zip (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>; result_selector: block<(l:TT;r:UU):auto>) : iterator<typedecl(result_selector(type<TT>,type<UU>))>*
- *zip (a: array<auto(TT)>; b: array<auto(UU)>; c: array<auto(WW)>) : array<tuple<TT;UU;WW>>*
- *zip (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>; var c: iterator<auto(WW)>) : iterator<tuple<TT;UU;WW>>*
- *zip_to_array (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>; var c: iterator<auto(WW)>) : array<tuple<TT;UU;WW>>*
- *zip_to_array (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>) : array<tuple<TT;UU>>*
- *zip_to_array (var a: iterator<auto(TT)>; var b: iterator<auto(UU)>; result_selector: block<(l:TT;r:UU):auto>) : array<typedecl(result_selector(type<TT>,type<UU>))>*

select

`linq::select(src: array<auto(TT)>) : array<tuple<int;TT>>()`

Projects each element of an array into a new form

Arguments

- **src** : array<auto(TT)>

`linq::select(src: iterator<auto(TT)>) : iterator<tuple<int;TT>>()`

`linq::select(src: iterator<auto(TT)>; result_selector: auto) : iterator<typedecl(result_selector(type<T`

`linq::select(src: array<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(type<TT>))>(`

---

select_many

`linq::select_many(src: iterator<auto(TT)>; result_selector: auto) : iterator<typedecl(result_selector(i`

Projects each element of an iterator to an iterator and flattens the resulting iterators into one iterator

Arguments

- **src** : iterator<auto(TT)>
- **result_selector** : auto

`linq::select_many(src: array<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(iter_ty`

`linq::select_many(src: array<auto(TT)>; collection_selector: auto; result_selector: auto) : array<typed`

`linq::select_many(src: iterator<auto(TT)>; collection_selector: auto; result_selector: auto) : iterator`

---

select_many_to_array

`linq::select_many_to_array(src: iterator<auto(TT)>; collection_selector: auto; result_selector: auto) :`

Projects each element of an iterator to an iterator and flattens the resulting iterators into one array

Arguments

- **src** : iterator<auto(TT)>
- **collection_selector** : auto
- **result_selector** : auto

`linq::select_many_to_array(src: iterator<auto(TT)>; result_selector: auto) : array<typedecl(result_sele`

---

select_to_array

linq::**select_to_array(src: iterator<auto(TT)>; result_selector: auto) : array<typedecl(result_selector(**

Projects each element of an iterator into a new form using a selector function and returns an array

Arguments

- **src** : iterator<auto(TT)>
- **result_selector** : auto

linq::**select_to_array(src: iterator<auto(TT)>) : array<tuple<int;TT>>()**

---

zip

linq::**zip(a: iterator<auto(TT)>; b: iterator<auto(UU)>) : iterator<tuple<TT;UU>>()**

Merges two iterators into an iterator of tuples

Arguments

- **a** : iterator<auto(TT)>
- **b** : iterator<auto(UU)>

linq::**zip(a: array<auto(TT)>; b: array<auto(UU)>) : array<tuple<TT;UU>>()**

linq::**zip(a: array<auto(TT)>; b: array<auto(UU)>; result_selector: block<(l:TT;r:UU):auto>) : array<typ**

linq::**zip(a: iterator<auto(TT)>; b: iterator<auto(UU)>; result_selector: block<(l:TT;r:UU):auto>) : ite**

linq::**zip(a: array<auto(TT)>; b: array<auto(UU)>; c: array<auto(WW)>) : array<tuple<TT;UU;WW>>()**

linq::**zip(a: iterator<auto(TT)>; b: iterator<auto(UU)>; c: iterator<auto(WW)>) : iterator<tuple<TT;UU;W**

---

zip_to_array

linq::**zip_to_array(a: iterator<auto(TT)>; b: iterator<auto(UU)>; c: iterator<auto(WW)>) : array<tuple<T**

Merges three iterators into an array of tuples

Arguments

- **a** : iterator<auto(TT)>
- **b** : iterator<auto(UU)>
- **c** : iterator<auto(WW)>

linq::**zip_to_array(a: iterator<auto(TT)>; b: iterator<auto(UU)>) : array<tuple<TT;UU>>()**

linq::**zip_to_array(a: iterator<auto(TT)>; b: iterator<auto(UU)>; result_selector: block<(l:TT;r:UU):aut**

6.3.12 Conversion operations

- *to_sequence (a: array<auto(TT)>) : iterator<TT>*
- *to_sequence_move (var a: array<auto(TT)>) : iterator<TT>*
- *to_table (a: array<auto(TT)>; key: block<(v:TT):auto>; elementSelector: block<(v:TT):auto>) : table<typedecl(_::unique_key(type<TT>)), typedecl(elementSelector(type<TT>))>*
- *to_table (var a: iterator<auto(TT)>; key: block<(v:TT):auto>; elementSelector: block<(v:TT):auto>) : table<typedecl(_::unique_key(type<TT>)), typedecl(elementSelector(type<TT>))>*

linq::**to_sequence(a: array<auto(TT)>) : iterator<TT>()**

Converts an array to an iterator

Arguments

- **a** : array<auto(TT)>

linq::**to_sequence_move(a: array<auto(TT)>) : iterator<TT>()**

Converts an array to an iterator, captures input

Arguments

- **a** : array<auto(TT)>

to_table

linq::**to_table(a: array<auto(TT)>; key: block<(v:TT):auto>; elementSelector: block<(v:TT):auto>) : tabl**

Converts an array to a table

Arguments

- **a** : array<auto(TT)>
- **key** : block<(v:TT):auto>
- **elementSelector** : block<(v:TT):auto>

linq::**to_table(a: iterator<auto(TT)>; key: block<(v:TT):auto>; elementSelector: block<(v:TT):auto>) : t**

6.3.13 Comparators and keys

- *less (a: tuple<auto(TT)>; b: tuple<auto(TT)>) : bool*
- *less (a: auto; b: auto) : bool*
- *less (a: tuple<auto(TT);auto(UU);auto(VV)>; b: tuple<auto(TT);auto(UU);auto(VV)>) : bool*
- *less (a: tuple<auto(TT);auto(UU)>; b: tuple<auto(TT);auto(UU)>) : bool*
- *less (a: tuple<auto(TT);auto(UU);auto(VV);auto(WW)>; b: tuple<auto(TT);auto(UU);auto(VV);auto(WW)>) : bool*
- *sequence_equal (var first: iterator<auto(TT)>; var second: iterator<auto(TT)>) : bool*
- *sequence_equal (first: array<auto(TT)>; second: array<auto(TT)>) : bool*
- *sequence_equal_by (var first: iterator<auto(TT)>; var second: iterator<auto(TT)>; key: block<(arg:TT):auto>) : bool*

- *sequence_equal_by (first: array<auto(TT)>; second: array<auto(TT)>; key: block<(arg:TT):auto>) : bool*

less

`linq::less(a: tuple<auto(TT)>; b: tuple<auto(TT)>) : bool()`

Compares two tuples, returns true if first is less than second

Arguments

- **a** : tuple<auto(TT)>
- **b** : tuple<auto(TT)>

`linq::less(a: auto; b: auto) : bool()`

`linq::less(a: tuple<auto(TT);auto(UU);auto(VV)>; b: tuple<auto(TT);auto(UU);auto(VV)>) : bool()`

`linq::less(a: tuple<auto(TT);auto(UU)>; b: tuple<auto(TT);auto(UU)>) : bool()`

`linq::less(a: tuple<auto(TT);auto(UU);auto(VV);auto(WW)>; b: tuple<auto(TT);auto(UU);auto(VV);auto(WW)>`

---

sequence_equal

`linq::sequence_equal(first: iterator<auto(TT)>; second: iterator<auto(TT)>) : bool()`

Checks if two sequences are equal

Arguments

- **first** : iterator<auto(TT)>
- **second** : iterator<auto(TT)>

`linq::sequence_equal(first: array<auto(TT)>; second: array<auto(TT)>) : bool()`

---

sequence_equal_by

`linq::sequence_equal_by(first: iterator<auto(TT)>; second: iterator<auto(TT)>; key: block<(arg:TT):auto`

Checks if two sequences are equal by key

Arguments

- **first** : iterator<auto(TT)>
- **second** : iterator<auto(TT)>
- **key** : block<(arg:TT):auto>

`linq::sequence_equal_by(first: array<auto(TT)>; second: array<auto(TT)>; key: block<(arg:TT):auto>) : b`

6.4 Boost module for LINQ

The LINQ_BOOST module extends LINQ with pipe-friendly macros using underscore syntax for inline predicates and selectors. Expressions like `arr |> _where(_ > 3) |> _select(_ * 2)` provide concise functional pipelines.

See also *LINQ* for the full set of query operations. See tutorial_linq for a hands-on tutorial.

All functions and symbols are in "linq_boost" module, use require to get access to it.

```
require daslib/linq_boost
```

Example:

```
require daslib/linq
    require daslib/linq_boost

    [export]
    def main() {
        var src <- [iterator for (x in range(10)); x]
        var evens <- _where(src, _ % 2 == 0)
        for (v in evens) {
            print("{v} ")
        }
        print("\n")
    }
    // output:
    // 0 2 4 6 8
```

6.4.1 Call macros

`linq_boost::`**`_order_by`**

implements _order_by(iterator, expression) shorthand notation that expands into order_by(iterator, $(_) => expression) for example:

```
each(foo)._order_by(_.id)
```

`linq_boost::`**`_union_by`**

implements _union_by(iterator1, iterator2, expression) shorthand notation that expands into union_by(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._union_by(each(foo2), _.id)
```

`linq_boost::`**`_take_while`**

implements _take_while(iterator, expression) shorthand notation that expands into take_while(iterator, $(_) => expression) for example:

```
each(foo)._take_while(_ < 5)
```

`linq_boost::`**`_all`**

implements _all(iterator, expression) shorthand notation that expands into all(iterator, $(_) => expression) for example:

```
each(foo)._all(_ < 5)
```

linq_boost::**_distinct_by_to_array**

implements _distinct_by_to_array(iterator, expression) shorthand notation that expands into distinct_by_to_array(iterator, $(_) => expression) for example:

```
each(foo)._distinct_by_to_array(_.id)
```

linq_boost::**_order_by_to_array**

implements _order_by_to_array(iterator, expression) shorthand notation that expands into order_by_to_array(iterator, $(_) => expression) for example:

```
each(foo)._order_by_to_array(_.id)
```

linq_boost::**_except_by**

implements _except_by(iterator1, iterator2, expression) shorthand notation that expands into except_by(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._except_by(each(foo2), _.id)
```

linq_boost::**_min_max_average_by**

implements _min_max_average_by(iterator, expression) shorthand notation that expands into min_max_average_by(iterator, $(_) => expression) for example:

```
each(foo)._min_max_average_by(_.value)
```

linq_boost::**_min_by**

implements _min_by(iterator, expression) shorthand notation that expands into min_by(iterator, $(_) => expression) for example:

```
each(foo)._min_by(_.value)
```

linq_boost::**_select_to_array**

implements _select_to_array(iterator, expression) shorthand notation that expands into select_to_array(iterator, $(_) => expression) for example:

```
each(foo)._select_to_array(_ * 2)
```

linq_boost::**_union_by_to_array**

implements _union_by_to_array(iterator1, iterator2, expression) shorthand notation that expands into union_by_to_array(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._union_by_to_array(each(foo2), _.id)
```

linq_boost::**_unique_by_to_array**

implements _unique_by_to_array(iterator, expression) shorthand notation that expands into unique_by_to_array(iterator, $(_) => expression) for example:

```
each(foo)._unique_by_to_array(_.id)
```

linq_boost::**_count**

implements _count(iterator, expression) shorthand notation that expands into count(iterator, $(_) => expression) for example:

```
each(foo)._count(_ > 3)
```

linq_boost::**_max_by**

implements _max_by(iterator, expression) shorthand notation that expands into max_by(iterator, $(_) => expression) for example:

```
each(foo)._max_by(_.value)
```

linq_boost::**_any**

implements _any(iterator, expression) shorthand notation that expands into any(iterator, $(_) => expression) for example:

```
each(foo)._any(_ < 5)
```

linq_boost::**_except_by_to_array**

implements _except_by_to_array(iterator1, iterator2, expression) shorthand notation that expands into except_by_to_array(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._except_by_to_array(each(foo2), _.id)
```

linq_boost::**_order_by_descending**

implements _order_by_descending(iterator, expression) shorthand notation that expands into order_by_descending(iterator, $(_) => expression) for example:

```
each(foo)._order_by_descending(_.id)
```

linq_boost::**_order_by_descending_to_array**

implements _order_by_descending_to_array(iterator, expression) shorthand notation that expands into order_by_descending_to_array(iterator, $(_) => expression) for example:

```
each(foo)._order_by_descending_to_array(_.id)
```

linq_boost::**_distinct_by**

implements _distinct_by(iterator, expression) shorthand notation that expands into distinct_by(iterator, $(_) => expression) for example:

```
each(foo)._distinct_by(_.id)
```

linq_boost::**_select**

implements _select(iterator, expression) shorthand notation that expands into select(iterator, $(_) => expression) for example:

```
each(foo)._select(_ * 2)
```

linq_boost::**_intersect_by_to_array**

implements _intersect_by_to_array(iterator1, iterator2, expression) shorthand notation that expands into intersect_by_to_array(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._intersect_by_to_array(each(foo2), _.id)
```

linq_boost::**_min_max_by**

implements _min_max_by(iterator, expression) shorthand notation that expands into min_max_by(iterator, $(_) => expression) for example:

```
each(foo)._min_max_by(_.value)
```

linq_boost::**_intersect_by**

implements _intersect_by(iterator1, iterator2, expression) shorthand notation that expands into intersect_by(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._intersect_by(each(foo2), _.id)
```

linq_boost::**_where_to_array**

implements _where_to_array(iterator, expression) shorthand notation that expands into where_to_array(iterator, $(_) => expression) for example:

```
each(foo)._where_to_array(_ < 5)
```

linq_boost::**_skip_while**

implements _skip_while(iterator, expression) shorthand notation that expands into skip_while(iterator, $(_) => expression) for example:

```
each(foo)._skip_while(_ < 5)
```

linq_boost::**_where**

implements _where(iterator, expression) shorthand notation that expands into where_(iterator, $(_) => expression) for example:

```
each(foo)._where(_ < 5)
```

linq_boost::**_unique_by**

implements _unique_by(iterator, expression) shorthand notation that expands into unique_by(iterator, $(_) => expression) for example:

```
each(foo)._unique_by(_.id)
```

linq_boost::**_fold**

implements _fold(expression) that folds LINQ expressions into optimized sequnences for example:

```
_fold(each(foo)._where(_ > 5)._select(_ * 2))
```

expands into a single comprehension that does all operations in one pass

linq_boost::**_sequence_equal_by**

implements _sequence_equal_by(iterator1, iterator2, expression) shorthand notation that expands into sequence_equal_by(iterator1, iterator2, $(_) => expression) for example:

```
each(foo1)._sequence_equal_by(each(foo2), _.id)
```

6.5 Pattern matching

The MATCH module implements pattern matching on variants, structs, tuples, arrays, and scalar values. Supports variable capture, wildcards, guard expressions, and alternation. `static_match` enforces exhaustive matching at compile time.

See tutorial_pattern_matching for a hands-on tutorial.

All functions and symbols are in "match" module, use require to get access to it.

```
require daslib/match
```

Example:

```
require daslib/match

    enum Color {
        red
        green
        blue
    }

    def describe(c : Color) : string {
        match (c) {
            if (Color.red) { return "red"; }
            if (Color.green) { return "green"; }
            if (_) { return "other"; }
        }
        return "?"
    }

    [export]
    def main() {
        print("{describe(Color.red)}\n")
        print("{describe(Color.green)}\n")
        print("{describe(Color.blue)}\n")
    }
    // output:
    // red
    // green
    // other
```

6.5.1 Call macros

`match::`**`match`**

Implements *match* macro.

`match::`**`static_match`**

Implements *static_match* macro.

`match::`**`multi_match`**

Implements *multi_match* macro.

`match::`**`static_multi_match`**

Implements *static_multi_match* macro.

6.5.2 Structure macros

`match::`**`match_as_is`**

Implements *match_as_is* annotation. This annotation is used to mark that structure can be matched with different type via is and as machinery.

`match::`**`match_copy`**

Implements *match_copy* annotation. This annotation is used to mark that structure can be matched with different type via match_copy machinery.

CHAPTER

SEVEN

DATA FORMATS

JSON, regular expressions, and reStructuredText processing.

7.1 JSON manipulation library

The JSON module implements JSON parsing and serialization. It provides `read_json` for parsing JSON text into a `JsonValue` tree, `write_json` for serializing back to text, and `JV` helpers for constructing JSON values from daslang types.

See also *Boost package for JSON* for automatic struct-to-JSON conversion and the `%json~` reader macro. See tutorial_json for a hands-on tutorial.

All functions and symbols are in "json" module, use require to get access to it.

```
require daslib/json
```

Example:

```
require daslib/json

    [export]
    def main() {
        let data = "[1, 2, 3]"
        var error = ""
        var js <- read_json(data, error)
        print("json: {write_json(js)}\n")
        unsafe {
            delete js
        }
    }
    // output:
    // json: [1,2,3]
```

7.1.1 Type aliases

`json::variant JsValue`

Single JSON element.

Variants

- **_object** : table<string; *JsonValue*?> - JSON object
- **_array** : array< *JsonValue*?> - JSON array
- **_string** : string - JSON string
- **_number** : double - JSON number
- **_longint** : int64 - extension, not part of JSON standard (represents long integer numbers)
- **_bool** : bool - JSON boolean
- **_null** : void? - JSON null

`json::variant Token`

JSON input stream token.

Variants

- **_string** : string - string token
- **_number** : double - number token
- **_longint** : int64 - extension, not part of JSON standard (represents long integer numbers)
- **_bool** : bool - boolean token
- **_null** : void? - null token
- **_symbol** : int - symbol token (one of []{}:,)
- **_error** : string - error token

7.1.2 Structures

`json::JsonValue`

JSON value, wraps any JSON element.

Fields

- **value** : *JsValue* - value of the JSON element

`json::TokenAt`

JSON parsing token. Contains token and its position.

Fields

- **value** : *Token* - token value
- **line** : int - token position in the input stream
- **row** : int - token position in the input stream

7.1.3 Value conversion

- *JV (v: string) : JsonValue?*
- *JV (v: double) : JsonValue?*
- *JV (v: bool) : JsonValue?*
- *JV (var v: table<string, JsonValue?>) : JsonValue?*
- *JV (v: float) : JsonValue?*
- *JV (var v: array<JsonValue?>) : JsonValue?*
- *JV (v: int) : JsonValue?*
- *JV (v: bitfield) : JsonValue?*
- *JV (v: bitfield16:uint16<>) : JsonValue?*
- *JV (v: bitfield8:uint8<>) : JsonValue?*
- *JV (v: bitfield64:uint64<>) : JsonValue?*
- *JV (val: int8) : JsonValue?*
- *JV (val: uint8) : JsonValue?*
- *JV (val: int16) : JsonValue?*
- *JV (val: uint) : JsonValue?*
- *JV (val: uint16) : JsonValue?*
- *JV (val: int64) : JsonValue?*
- *JV (val: uint64) : JsonValue?*
- *JVNull () : JsonValue?*

JV

`json::JV(v: string) : JsonValue?()`

Creates *JsonValue* out of string value.

Arguments

- **v** : string

`json::JV(v: double) : JsonValue?()`

`json::JV(v: bool) : JsonValue?()`

`json::JV(v: table<string, JsonValue?>) : JsonValue?()`

`json::JV(v: float) : JsonValue?()`

`json::JV(v: array<JsonValue?>) : JsonValue?()`

`json::JV(v: int) : JsonValue?()`

`json::JV(v: bitfield) : JsonValue?()`

`json::JV(v: bitfield16:uint16<>) : JsonValue?()`

json::**JV(v: bitfield8:uint8<>) : JsonValue?**()

json::**JV(v: bitfield64:uint64<>) : JsonValue?**()

json::**JV(val: int8) : JsonValue?**()

json::**JV(val: uint8) : JsonValue?**()

json::**JV(val: int16) : JsonValue?**()

json::**JV(val: uint) : JsonValue?**()

json::**JV(val: uint16) : JsonValue?**()

json::**JV(val: int64) : JsonValue?**()

json::**JV(val: uint64) : JsonValue?**()

---

json::**JVNull() : JsonValue?**()

Creates *JsonValue* representing *null*.

7.1.4 Read and write

- *read_json (text: array<uint8>; var error: string&) : JsonValue?*
- *read_json (text: string; var error: string&) : JsonValue?*
- *write_json (val: JsonValue?#) : string*
- *write_json (val: JsonValue?) : string*

read_json

json::**read_json(text: array<uint8>; error: string&) : JsonValue?**()

reads JSON from the *text* array of uint8. if *error* is not empty, it contains the parsing error message.

Arguments

- **text** : array<uint8>
- **error** : string&

json::**read_json(text: string; error: string&) : JsonValue?**()

---

write_json

`json::write_json(val: JsonValue?#) : string()`

Overload accepting temporary type

Arguments

- **val** : *JsonValue?#*

`json::write_json(val: JsonValue?) : string()`

7.1.5 JSON properties

- *set_allow_duplicate_keys (value: bool) : bool*
- *set_no_empty_arrays (value: bool) : bool*
- *set_no_trailing_zeros (value: bool) : bool*

`json::set_allow_duplicate_keys(value: bool) : bool()`

if *value* is true, then duplicate keys are allowed in objects. the later key overwrites the earlier one.

Arguments

- **value** : bool

`json::set_no_empty_arrays(value: bool) : bool()`

if *value* is true, then empty arrays are not written at all

Arguments

- **value** : bool

`json::set_no_trailing_zeros(value: bool) : bool()`

if *value* is true, then numbers are written without trailing zeros.

Arguments

- **value** : bool

7.1.6 Broken JSON

- *try_fixing_broken_json (var bad: string) : string*

`json::try_fixing_broken_json(bad: string) : string()`

fixes broken json. so far supported 1. “string” + “string” string concatenation 2. “text “nested text” text” nested quotes 3. extra , at the end of object or array 4. /uXXXXXX sequences in the middle of white space

Arguments

- **bad** : string

7.2 Boost package for JSON

The JSON_BOOST module extends JSON support with operator overloads for convenient field access (`?[]`), null-coalescing (`??`), and automatic struct-to-JSON conversion macros (`from_JsValue`, `to_JsValue`).

See also *JSON manipulation library* for core JSON parsing and writing. See tutorial_json for a hands-on tutorial.

All functions and symbols are in "json_boost" module, use require to get access to it.

```
require daslib/json_boost
```

Example:

```
require daslib/json_boost

    [export]
    def main() {
        let data = "\{ \"name\": \"Alice\", \"age\": 30 \}"
        var error = ""
        var js <- read_json(data, error)
        if (error == "") {
            let name = js?.name ?? "?"
            print("name = {name}\n")
            let age = js?.age ?? -1
            print("age = {age}\n")
        }
        unsafe {
            delete js
        }
    }
    // output:
    // name = Alice
    // age = 30
```

7.2.1 Field annotations

Struct fields can carry annotations that control how `JV` / `from_JV` and the builtin `sprint_json` serialize and deserialize them. Annotations are parsed by *parse_json_annotation* into a *JsonFieldState* and stored in a `static_let` cache so each field is parsed only once.

`sprint_json` requires `options rtti` for annotations to take effect at runtime.

Annotation	Effect
`@optional`	Skip the field when its value is default / empty (`0`, `false`, empty string, empty array, empty table, null pointer).
`@rename="json_key"`	Use *json_key* instead of the daslang field name in JSON output and when looking up keys during `from_JV` deserialization. The annotation value must be a string (`@rename="name"`). A bare `@rename` with no string value is silently ignored.
`@embed`	Treat a `string` field as raw JSON — embed it without extra quoting. During `JV` conversion the string is parsed with `read_json` and the resulting sub-tree is inserted directly.
`@unescape`	Write the string field without escaping special characters (backslashes, quotes, etc.).
`@enum_as_int`	Serialize an enum field as its integer value instead of the enumeration name string.

Example with `sprint_json`:

```
options rtti

struct Config {
    name : string
    @optional debug : bool          // omitted when false
    @rename="type" _type : string   // JSON key is "type"
    @embed raw : string             // embedded as raw JSON
    @unescape path : string         // no escaping of backslashes
    @enum_as_int level : Priority   // integer, not string
}

let json_str = sprint_json(cfg, false)
```

See tutorial_json for runnable examples of every annotation.

Structures

`json_boost::`**`JsonFieldState`**

Per-field serialization options for JSON struct conversion.

Fields

- **argName** : string - name of the field in JSON
- **enumAsInt** : bool - whether to parse enum as integer
- **unescape** : bool - whether to unescape strings
- **embed** : bool - whether to embed the field
- **optional** : bool - whether the field is optional

Reader macros

`json_boost::`**`json`**

This macro implements embedding of the JSON object into the program:

```
var jsv = %json~
{
  "name": "main_window",
  "value": 500,
  "size": [1,2,3]
} %%
```

Variant macros

`json_boost::`**`better_json`**

This macro is used to implement *is json_value* and *as json_value* runtime checks. It essentially substitutes *value as name* with *value.value as name* and *value is name* with *value.value is name*.

Value conversion

- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8: auto; val9: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8: auto; val9: auto; val10: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto) : JsonValue?*
- *JV (val1: auto; val2: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8: auto) : JsonValue?*
- *JV (val1: auto; val2: auto; val3: auto) : JsonValue?*
- *JV (v: auto(VecT)) : auto*
- *JV (value: auto(TT)) : JsonValue?*
- *from_JV (v: JsonValue const?; ent: uint16; defV: uint16 = uint16(0)) : auto*
- *from_JV (v: JsonValue const?; ent: uint8; defV: uint8 = uint8(0)) : auto*
- *from_JV (v: JsonValue const?; ent: int8; defV: int8 = int8(0)) : auto*
- *from_JV (v: JsonValue const?; ent: int16; defV: int16 = int16(0)) : auto*
- *from_JV (v: JsonValue const?; ent: bitfield; defV: bitfield = bitfield()) : auto*
- *from_JV (v: JsonValue const?; ent: uint; defV: uint = 0x0) : auto*
- *from_JV (v: JsonValue const?; ent: int; defV: int = 0) : auto*
- *from_JV (v: JsonValue const?; ent: bitfield16:uint16<>; defV: bitfield16 = bitfield16()) : auto*
- *from_JV (v: JsonValue const?; ent: auto(VecT); defV: VecT = VecT()) : auto*
- *from_JV (v: JsonValue const?; ent: bitfield64:uint64<>; defV: bitfield64 = bitfield64()) : auto*
- *from_JV (v: JsonValue const?; anything: auto(TT)) : auto*
- *from_JV (v: JsonValue const?; anything: table<auto(KT), auto(VT)>) : auto*
- *from_JV (v: JsonValue const?; ent: int64; defV: int64 = 0) : auto*
- *from_JV (v: JsonValue const?; ent: float; defV: float = 0f) : auto*
- *from_JV (v: JsonValue const?; ent: string; defV: string = “”) : auto*
- *from_JV (v: JsonValue const?; ent: auto(EnumT); defV: EnumT = EnumT()) : EnumT*
- *from_JV (v: JsonValue const?; ent: bool; defV: bool = false) : auto*

- *from_JV (v: JsonValue const?; ent: double; defV: double = 0lf) : auto*
- *from_JV (v: JsonValue const?; ent: uint64; defV: uint64 = 0x0) : auto*
- *from_JV (v: JsonValue const?; ent: bitfield8:uint8<>; defV: bitfield8 = bitfield8()) : auto*

JV

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8`

Creates array of nine JsonValues.

Arguments

- **val1** : auto
- **val2** : auto
- **val3** : auto
- **val4** : auto
- **val5** : auto
- **val6** : auto
- **val7** : auto
- **val8** : auto
- **val9** : auto

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto) : JsonValue?()`

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto) : Js`

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8`

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto) : JsonValue?()`

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto) : JsonValue?()`

`json_boost::JV(val1: auto; val2: auto) : JsonValue?()`

`json_boost::JV(val1: auto; val2: auto; val3: auto; val4: auto; val5: auto; val6: auto; val7: auto; val8`

`json_boost::JV(val1: auto; val2: auto; val3: auto) : JsonValue?()`

`json_boost::JV(v: auto(VecT)) : auto()`

`json_boost::JV(value: auto(TT)) : JsonValue?()`

from_JV

json_boost::**from_JV(v: JsonValue const?; ent: uint16; defV: uint16 = uint16(0)) : auto()**

Parse a JSON value and return the corresponding native value.

Arguments

- **v** : *JsonValue*?
- **ent** : uint16
- **defV** : uint16

json_boost::**from_JV(v: JsonValue const?; ent: uint8; defV: uint8 = uint8(0)) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: int8; defV: int8 = int8(0)) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: int16; defV: int16 = int16(0)) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: bitfield; defV: bitfield = bitfield()) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: uint; defV: uint = 0x0) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: int; defV: int = 0) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: bitfield16:uint16<>; defV: bitfield16 = bitfield16()) : a**

json_boost::**from_JV(v: JsonValue const?; ent: auto(VecT); defV: VecT = VecT()) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: bitfield64:uint64<>; defV: bitfield64 = bitfield64()) : a**

json_boost::**from_JV(v: JsonValue const?; anything: auto(TT)) : auto()**

json_boost::**from_JV(v: JsonValue const?; anything: table<auto(KT), auto(VT)>) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: int64; defV: int64 = 0) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: float; defV: float = 0f) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: string; defV: string = "") : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: auto(EnumT); defV: EnumT = EnumT()) : EnumT()**

json_boost::**from_JV(v: JsonValue const?; ent: bool; defV: bool = false) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: double; defV: double = 0lf) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: uint64; defV: uint64 = 0x0) : auto()**

json_boost::**from_JV(v: JsonValue const?; ent: bitfield8:uint8<>; defV: bitfield8 = bitfield8()) : auto(**

Element access operators

- *JsonValue const? ==const?. (a: JsonValue const? ==const; key: string) : JsonValue?*
- *JsonValue const? ==const?[] (a: JsonValue const? ==const; key: string) : JsonValue?*
- *JsonValue const? ==const?[] (a: JsonValue const? ==const; idx: int) : JsonValue?*
- *JsonValue? ==const?. (var a: JsonValue? ==const; key: string) : JsonValue?*
- *JsonValue? ==const?[] (var a: JsonValue? ==const; idx: int) : JsonValue?*
- *JsonValue? ==const?[] (var a: JsonValue? ==const; key: string) : JsonValue?*

`JsonValue const? ==const?.(a: JsonValue const? ==const; key: string) : JsonValue?()`

Returns the value of the key in the JSON object, if it exists.

Arguments

- **a** : *JsonValue*?!
- **key** : string

JsonValue const? ==const?[]

`json_boost::JsonValue const? ==const?[](a: JsonValue const? ==const; key: string) : JsonValue?()`

Returns the value of the key in the JSON object, if it exists.

Arguments

- **a** : *JsonValue*?!
- **key** : string

`json_boost::JsonValue const? ==const?[](a: JsonValue const? ==const; idx: int) : JsonValue?()`

`JsonValue? ==const?.(a: JsonValue? ==const; key: string) : JsonValue?()`

Returns the value of the key in the JSON object, if it exists.

Arguments

- **a** : *JsonValue*?!
- **key** : string

JsonValue? ==const?[]

`json_boost::JsonValue? ==const?[](a: JsonValue? ==const; idx: int) : JsonValue?()`

Returns the value of the index in the JSON array, if it exists.

Arguments

- **a** : *JsonValue*?!
- **idx** : int

`json_boost::JsonValue? ==const?[](a: JsonValue? ==const; key: string) : JsonValue?()`

Null coalescing operators

- *JsonValue const??? (a: JsonValue const?; val: float) : float*
- *JsonValue const??? (a: JsonValue const?; val: double) : double*
- *JsonValue const??? (a: JsonValue const?; val: int16) : int16*
- *JsonValue const??? (a: JsonValue const?; val: int8) : int8*
- *JsonValue const??? (a: JsonValue const?; val: uint8) : uint8*
- *JsonValue const??? (a: JsonValue const?; val: int64) : int64*
- *JsonValue const??? (a: JsonValue const?; val: uint16) : uint16*
- *JsonValue const??? (a: JsonValue const?; val: int) : int*
- *JsonValue const??? (a: JsonValue const?; val: bool) : bool*
- *JsonValue const??? (a: JsonValue const?; val: uint64) : uint64*
- *JsonValue const??? (a: JsonValue const?; val: uint) : uint*
- *JsonValue const??? (a: JsonValue const?; val: string) : string*

JsonValue const???

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: float) : float`**`()`

Returns the value of the JSON object, if it exists, otherwise returns the default value.

Arguments

- **a** : *JsonValue*?
- **val** : float

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: double) : double`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: int16) : int16`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: int8) : int8`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: uint8) : uint8`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: int64) : int64`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: uint16) : uint16`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: int) : int`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: bool) : bool`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: uint64) : uint64`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: uint) : uint`**`()`

`json_boost::`**`JsonValue const???(a: JsonValue const?; val: string) : string`**`()`

Value extraction

- *JsonValue const? ==const?.`value (a: JsonValue const? ==const) : variant<_object:table<string;JsonValue?>;_array:array<JsonValue?>;_string:string;_number:double;_longint:int64;_bool:bool;_ const?*
- *JsonValue? ==const?.`value (var a: JsonValue? ==const) : variant<_object:table<string;JsonValue?>;_array:array<JsonValue?>;_string:string;_number:double;_longint:int64;_bool:bool;_*

`JsonValue const? ==const?.`**`` `value(a: JsonValue const? ==const) : variant<_object:table<string;JsonValue? ``**

Returns the value of the JSON object, if it exists.

Arguments

- **a** : *JsonValue*?!

`JsonValue? ==const?.`**`` `value(a: JsonValue? ==const) : variant<_object:table<string;JsonValue?>;_array:arr ``**

Returns the value of the JSON object, if it exists.

Arguments

- **a** : *JsonValue*?!

Annotation parsing

- *parse_json_annotation (name: string; annotation: array<tuple<name:string;data:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:i : JsonFieldState*

`json_boost::`**`parse_json_annotation(name: string; annotation: array<tuple<name:string;data:variant<tBool:`**

Parse JSON field annotations and return the corresponding JsonFieldState.

Arguments

- **name** : string
- **annotation** : array<tuple<name:string;data: *RttiValue*>>

7.3 Regular expression library

The REGEX module implements regular expression matching and searching. It provides `regex_compile` for building patterns, `regex_match` for full-string matching, `regex_search` for finding the first match anywhere, `regex_foreach` for iterating all matches, `regex_replace` for substitution (both block-based and template-string forms), `regex_split` for splitting strings, `regex_match_all` for collecting all match ranges, `regex_group` for capturing groups by index, and `regex_group_by_name` for named group lookup.

See tutorial_regex for a hands-on tutorial.

Supported syntax:

- `.` — any character except newline (use `dot_all=true` to also match `\n`)
- `^` — beginning of string (or offset position)
- `$` — end of string
- `+` — one or more (greedy)
- `*` — zero or more (greedy)

- `?` — zero or one (greedy)
- `+?` — one or more (lazy)
- `*?` — zero or more (lazy)
- `??` — zero or one (lazy)
- `{n}` — exactly *n* repetitions
- `{n,}` — *n* or more (greedy)
- `{n,m}` — between *n* and *m* (greedy)
- `{n}?` `{n,}?` `{n,m}?` — counted repetitions (lazy)
- `(...)` — capturing group
- `(?:...)` — non-capturing group
- `(?P<name>...)` — named capturing group
- `(?=...)` — positive lookahead assertion
- `(?!...)` — negative lookahead assertion
- `|` — alternation
- `[abc]`, `[a-z]`, `[^abc]` — character sets (negated with `^`)
- `\w` `\W` — word / non-word characters
- `\d` `\D` — digit / non-digit characters
- `\s` `\S` — whitespace / non-whitespace characters
- `\b` `\B` — word boundary / non-boundary assertions
- `\t` `\n` `\r` `\f` `\v` — whitespace escapes
- `\xHH` — hexadecimal character escape
- `\.` `\+` `\*` `\(` `\)` `\[` `\]` `\|` `\\` `\^` `\{` `\}` — escaped metacharacters

Flags:

- `case_insensitive=true` — ASCII case-insensitive matching (pass to `regex_compile`)
- `dot_all=true` — `.` also matches `\n` (pass to `regex_compile`)

Template-string replacement:

`regex_replace(re, str, replacement)` replaces matches using a template string. Supported references: `$0` or `$&` for the whole match, `$1`–`$9` for numbered groups, `${name}` for named groups, `$$` for a literal `$`.

The engine is ASCII-only (256-bit `CharSet`). Matching is anchored — `regex_match` tests from position 0 (or the given offset) and does NOT search; use `regex_search` to find the first occurrence, or `regex_foreach` / `regex_match_all` to find all occurrences.

See also *Boost package for REGEX* for compile-time regex construction via the `%regex~` reader macro.

All functions and symbols are in "regex" module, use require to get access to it.

```
require daslib/regex
```

Example:

```
require daslib/regex
    require strings

    [export]
    def main() {
        var re <- regex_compile("[0-9]+")
        let m = regex_match(re, "123abc")
        print("match length = {m}\n")
        let text = "age 25, height 180"
        regex_foreach(re, text) $(r) {
            print("found: {slice(text, r.x, r.y)}\n")
            return true
        }
    }
    // output:
    // match length = 3
    // found: 25
    // found: 180
```

7.3.1 Type aliases

`regex::CharSet = uint[8]`

Bitfield character set used internally by the regex engine.

`regex::ReGenRandom = iterator<uint>`

Random number generator callback used by `re_gen` for regex-based string generation.

`regex::variant MaybeReNode`

Regex node or nothing.

Variants

- **value** : *ReNode*? - Node.
- **nothing** : void? - Nothing.

7.3.2 Enumerations

`regex::ReOp`

Type of regular expression operation.

Values

- **Char** = 0 - Matching a character
- **Set** = 1 - Matching a character set
- **Any** = 2 - Matches any character
- **Eos** = 3 - Matches end of string
- **Bos** = 4 - Matches beginning of string
- **Group** = 5 - Matching a group

- **Plus** = 6 - Repetition: one or more
- **Star** = 7 - Repetition: zero or more
- **Question** = 8 - Repetition: zero or one
- **Concat** = 9 - First followed by second
- **Union** = 10 - Either first or second
- **Repeat** = 11 - Counted repetition: {n}, {n,}, {n,m}
- **WordBoundary** = 12 - Matches at a word boundary
- **NonWordBoundary** = 13 - Matches at a non-word boundary
- **Lookahead** = 14 - Positive lookahead assertion (?=…)
- **NegativeLookahead** = 15 - Negative lookahead assertion (?!…)

7.3.3 Structures

`regex::`**`ReNode`**

Regular expression node.

Fields

- **op** : *ReOp* - Regex operation
- **id** : int - Unique node identifier
- **fun2** : function<(regex: *Regex*;node: *ReNode*?;str:uint8?):uint8?> - Matchig function
- **gen2** : function<(node: *ReNode*?;rnd: *ReGenRandom*;str: *StringBuilderWriter*):void> - Generator function
- **at** : range - Source range
- **text** : string - Text fragment
- **textLen** : int - Length of text fragment
- **all** : array< *ReNode*?> - All child nodes
- **left** : *ReNode*? - Left child node
- **right** : *ReNode*? - Right child node
- **subexpr** : *ReNode*? - Subexpression node
- **next** : *ReNode*? - Next node in the list
- **cset** : *CharSet* - Character set for character class matching
- **index** : int - Index for character class matching
- **min_rep** : int - Minimum repetition count for counted quantifiers
- **max_rep** : int - Maximum repetition count for counted quantifiers (-1 means unlimited)
- **lazy** : bool - Whether this quantifier uses lazy matching (`*?`, `+?`, `??`, `{n,m}?`)
- **tail** : uint8? - Tail of the string

`regex::`**`Regex`**

Regular expression structure.

Fields

- **root** : *ReNode*? - Root node of the regex.
- **match** : uint8? - Original source text.
- **groups** : array<tuple<range;string>> - Captured groups.
- **earlyOut** : *CharSet* - Character set for early out optimization.
- **canEarlyOut** : bool - Whether early out optimization is enabled.
- **caseInsensitive** : bool - When true, matching is case-insensitive (ASCII only).
- **dotAll** : bool - When true, `.` matches newline characters as well.

7.3.4 Compilation and validation

- *debug_set (cset: CharSet)*
- *is_valid (var re: Regex) : bool*
- *regex_compile (expr: string; case_insensitive: bool = false; dot_all: bool = false) : Regex*
- *regex_compile (var re: Regex; expr: string; case_insensitive: bool = false; dot_all: bool = false) : bool*
- *regex_compile (var re: Regex) : Regex*
- *regex_debug (regex: Regex)*
- *visit_top_down (var node: ReNode?; blk: block<(var n:ReNode?):void>)*

`regex::`**`debug_set`**`(`*cset: CharSet*`)`

Prints all characters contained in a `CharSet` for debugging purposes.

Arguments

- **cset** : *CharSet*

`regex::`**`is_valid(re: Regex) : bool`**`()`

Returns `true` if the compiled regex is valid and ready for matching.

Arguments

- **re** : *Regex*

regex_compile

`regex::`**`regex_compile(expr: string; case_insensitive: bool = false; dot_all: bool = false) : Regex`**`()`

Compiles a regular expression pattern string into a `Regex` object. Panics if the pattern is invalid. An overload taking a `var re : Regex` out-parameter returns `bool` instead of panicking. Optional flags: `case_insensitive=true` for ASCII case-insensitive matching, `dot_all=true` for `.` to also match newline characters.

Arguments

- **expr** : string
- **case_insensitive** : bool

- **dot_all** : bool

regex::**regex_compile(re: Regex; expr: string; case_insensitive: bool = false; dot_all: bool = false) :**

regex::**regex_compile(re: Regex) : Regex**()

---

regex::**regex_debug**(*regex: Regex*)

Prints the internal structure of a compiled regex for debugging purposes.

Arguments

- **regex** : *Regex*

regex::**visit_top_down**(*node: ReNode?; blk: block<(var n:ReNode?):void>*)

Visits all nodes of a compiled regex tree in top-down order, invoking a callback for each node.

Arguments

- **node** : *ReNode*?
- **blk** : block<(n: *ReNode*?):void>

7.3.5 Access

- *Regex[] (regex: Regex; index: int) : range*
- *Regex[] (regex: Regex; name: string) : range*
- *regex_foreach (var regex: Regex; str: string; blk: block<(at:range):bool>)*
- *regex_group (regex: Regex; index: int; match: string) : string*
- *regex_group_by_name (regex: Regex; name: string; str: string) : string*

Regex[]

regex::**Regex[](regex: Regex; index: int) : range**()

Returns the match range for the given group index.

Arguments

- **regex** : *Regex*
- **index** : int

regex::**Regex[](regex: Regex; name: string) : range**()

---

regex::**regex_foreach**(*regex: Regex; str: string; blk: block<(at:range):bool>*)

Iterates over all non-overlapping matches of a regex in a string, invoking a block for each match.

Arguments

- **regex** : *Regex*
- **str** : string

- **blk** : block<(at:range):bool>

`regex::`**`regex_group(regex: Regex; index: int; match: string) : string`**`()`

Returns the substring captured by the specified group index after a successful match.

Arguments

- **regex** : *Regex*
- **index** : int
- **match** : string

`regex::`**`regex_group_by_name(regex: Regex; name: string; str: string) : string`**`()`

Returns the matched substring for the named capturing group `(?P<name>...)`. Returns empty string if the group name is not found.

Arguments

- **regex** : *Regex*
- **name** : string
- **str** : string

7.3.6 Match & replace

- *regex_match (var regex: Regex; str: string; offset: int = 0) : int*
- *regex_match_all (var regex: Regex; str: string) : array<range>*
- *regex_replace (var regex: Regex; str: string; blk: block<(at:string):string>) : string*
- *regex_replace (var regex: Regex; str: string; replacement: string) : string*
- *regex_search (var regex: Regex; str: string; offset: int = 0) : int2*
- *regex_split (var regex: Regex; str: string) : array<string>*

`regex::`**`regex_match(regex: Regex; str: string; offset: int = 0) : int`**`()`

Matches a compiled regex against a string and returns the end position of the match, or `-1` on failure.

Arguments

- **regex** : *Regex*
- **str** : string
- **offset** : int

`regex::`**`regex_match_all(regex: Regex; str: string) : array<range>`**`()`

Returns an array of all non-overlapping match ranges for the regular expression in `str`.

Arguments

- **regex** : *Regex*
- **str** : string

regex_replace

regex::**regex_replace(regex: Regex; str: string; blk: block<(at:string):string>) : string**()

Replaces each substring matched by the regex with the result returned by the provided block. An overload accepting a template string is also available, supporting `$0`/`$&` for the whole match, `$1`–`$9` for numbered groups, `${name}` for named groups, and `$$` for a literal `$`.

Arguments

- **regex** : *Regex*
- **str** : string
- **blk** : block<(at:string):string>

regex::**regex_replace(regex: Regex; str: string; replacement: string) : string**()

---

regex::**regex_search(regex: Regex; str: string; offset: int = 0) : int2**()

Searches for the first occurrence of the regular expression anywhere in `str`, starting from `offset`. Returns `int2(start, end)` on success, or `int2(-1, -1)` if not found. Unlike `regex_match`, this function scans the entire string.

Arguments

- **regex** : *Regex*
- **str** : string
- **offset** : int

regex::**regex_split(regex: Regex; str: string) : array<string>**()

Splits `str` by all non-overlapping matches of the regular expression. Returns an array of substrings between matches.

Arguments

- **regex** : *Regex*
- **str** : string

7.3.7 Generation

- *re_gen (var re: Regex; var rnd: ReGenRandom) : string*
- *re_gen_get_rep_limit () : uint*

regex::**re_gen(re: Regex; rnd: ReGenRandom) : string**()

Generates a random string that matches the given compiled regex.

Arguments

- **re** : *Regex*
- **rnd** : *ReGenRandom*

regex::**re_gen_get_rep_limit() : uint**()

Returns the maximum repetition limit used by regex quantifiers during string generation.

7.4 Boost package for REGEX

The REGEX_BOOST module extends regular expressions with the `%regex~` reader macro for compile-time regex construction. Inside the reader macro, backslashes are literal — no double-escaping is needed (e.g. `%regex~\d{3}%%` instead of `"\\d\{3}"`).

Optional flags can be appended after a second ~ separator:

- `%regex~pattern~i%%` — case-insensitive matching
- `%regex~pattern~s%%` — dot-all mode (`.` matches `\n`)
- `%regex~pattern~is%%` — both flags combined

See *Regular expression library* for the full list of supported syntax. See tutorial_regex for a hands-on tutorial.

All functions and symbols are in "regex_boost" module, use require to get access to it.

```
require daslib/regex_boost
```

Example:

```
require daslib/regex_boost
    require strings

    [export]
    def main() {
        var inscope re <- %regex~\d+%%
        let m = regex_match(re, "123abc")
        print("match length = {m}\n")
        let text = "age 25, height 180"
        regex_foreach(re, text) $(r) {
            print("found: {slice(text, r.x, r.y)}\n")
            return true
        }
    }
    // output:
    // match length = 3
    // found: 25
    // found: 180
```

7.4.1 Reader macros

regex_boost::**regex**

Reader macro that converts `%regex~` literals into precompiled `regex::Regex` objects at compilation time. Optional flags can follow a second ~: `%regex~pattern~i%%` for case-insensitive, `%regex~pattern~s%%` for dot-all, `%regex~pattern~is%%` for both.

7.5 Documentation generator

The RST module implements the documentation generation pipeline for daslang. It uses RTTI to introspect modules, types, and functions, then produces reStructuredText output suitable for Sphinx documentation builds.

All functions and symbols are in "rst" module, use require to get access to it.

```
require daslib/rst
```

7.5.1 Structures

`rst::`**`DocGroup`**

Group of documentation items.

Fields

- **name** : string - Name of the group.
- **func** : array<tuple<fn: *Function*?;mod: *Module*?>> - Functions in the group.
- **hidden** : bool - Whether the group is hidden.
- **_module** : *Module*? - Module, to which this group belongs.

`rst::`**`DocsHook`**

Hook for RST documentation generation.

Fields

- **annotationFilter** : lambda<(ann: *Annotation*):bool> - Filter for the supported annotations.
- **afterEnums** : lambda<(f: *FILE*?;was_enums:bool):void> - Additional generation hook after the enumerations.

7.5.2 Document writers

- *document (name: string; var mod: Module?; fname: string; var groups: array<DocGroup>; hook: DocsHook = DocsHook())*
- *document_enumeration (doc_file: file; mod: Module?; value: auto) : auto*
- *document_enumerations (doc_file: file; mods: array<Module?>) : bool*
- *documents (name: string; mods: array<Module?>; fname: string; var groups: array<DocGroup>; var hook: DocsHook = DocsHook())*

`rst::`**`document`**(*name: string; mod: Module?; fname: string; groups: array<DocGroup>; hook: DocsHook = DocsHook()*)

Generates RST documentation for a single module and writes it to a file.

Arguments

- **name** : string
- **mod** : *Module*?
- **fname** : string

- **groups** : array< *DocGroup*>
- **hook** : *DocsHook*

`rst::`**`document_enumeration(doc_file: file; mod: Module?; value: auto) : auto`**`()`

Generates RST documentation for a single enumeration type.

Arguments

- **doc_file** : *file*
- **mod** : *Module*?
- **value** : auto

`rst::`**`document_enumerations(doc_file: file; mods: array<Module?>) : bool`**`()`

Generates RST documentation for all enumerations in the given modules.

Arguments

- **doc_file** : *file*
- **mods** : array< *Module*?>

`rst::`**`documents`**(*name: string; mods: array<Module?>; fname: string; groups: array<DocGroup>; hook: DocsHook = DocsHook()*)

Generates RST documentation for multiple modules and writes them to files.

Arguments

- **name** : string
- **mods** : array< *Module*?>
- **fname** : string
- **groups** : array< *DocGroup*>
- **hook** : *DocsHook*

7.5.3 Descriptions

- *describe_short (expr: Expression?|smart_ptr<Expression>) : auto*

`rst::`**`describe_short(expr: Expression?|smart_ptr<Expression>) : auto`**`()`

Returns a concise one-line description of an expression or type.

Arguments

- **expr** : option< *Expression*?| smart_ptr< *Expression*>&>

7.5.4 Label makers

- *function_label_file (name: auto; value: smart_ptr<TypeDecl>; drop_args: int = 0) : auto*
- *function_label_file (value: smart_ptr<Function>|Function?; drop_args: int = 0) : auto*

function_label_file

`rst::`**`function_label_file(name: auto; value: smart_ptr<TypeDecl>; drop_args: int = 0) : auto`**`()`

Creates a unique, file-name-safe label string for a function.

Arguments

- **name** : auto
- **value** : smart_ptr< *TypeDecl*>&
- **drop_args** : int

`rst::`**`function_label_file(value: smart_ptr<Function>|Function?; drop_args: int = 0) : auto`**`()`

7.5.5 RST section makers

- *make_group (name: string; plus: string = "+") : string*

`rst::`**`make_group(name: string; plus: string = "+") : string`**`()`

Creates a named documentation group with a decorative RST section header.

Arguments

- **name** : string
- **plus** : string

7.5.6 Group operations

- *append_to_group_by_regex (var group: DocGroup; var mod: Module?; var reg: Regex) : DocGroup&*
- *group_by_regex (name: string; var mod: Module?; var reg: Regex) : DocGroup*
- *group_by_regex (name: string; var mods: array<Module?>; var reg: Regex) : DocGroup*
- *hide_group (var group: DocGroup) : DocGroup*

`rst::`**`append_to_group_by_regex(group: DocGroup; mod: Module?; reg: Regex) : DocGroup&`**`()`

Appends functions whose names match a regex to an existing documentation group.

Arguments

- **group** : *DocGroup*
- **mod** : *Module*?
- **reg** : *Regex*

group_by_regex

rst::**group_by_regex(name: string; mod: Module?; reg: Regex) : DocGroup**()

Groups module items whose names match the provided regular expression under a documentation section.

Arguments

- **name** : string
- **mod** : *Module*?
- **reg** : *Regex*

rst::**group_by_regex(name: string; mods: array<Module?>; reg: Regex) : DocGroup**()

---

rst::**hide_group(group: DocGroup) : DocGroup**()

Marks the specified documentation group as hidden so it is excluded from output.

Arguments

- **group** : *DocGroup*

7.5.7 Naming helpers

- *safe_function_name (name: string) : string*

rst::**safe_function_name(name: string) : string**()

Escapes special characters in a function name to produce a safe identifier for RST output.

Arguments

- **name** : string

CHAPTER

EIGHT

ENTITY COMPONENT SYSTEM

The decs (daslang Entity Component System) framework for data-oriented game architecture.

8.1 DECS, Daslang entity component system

The DECS module implements a Data-oriented Entity Component System. Entities are identified by integer IDs and store components as typed data. Systems query and process entities by their component signatures, enabling cache-friendly batch processing of game objects.

See tutorial_decs for a hands-on tutorial.

All functions and symbols are in “decs” module, use require to get access to it.

```
require daslib/decs
```

8.1.1 Type aliases

decs::**ComponentHash = uint64**

Hash value of the ECS component type

decs::**TypeHash = uint64**

Hash value of the individual type

decs::**DeferEval = lambda<(var act:DeferAction):void>**

Lambda which holds deferred action. Typically creation of destruction of an entity.

decs::**ComponentMap = array<ComponentValue>**

Table of component values for individual entity.

decs::**PassFunction = function<void>**

One of the callbacks which form individual pass.

8.1.2 Constants

`decs::INVALID_ENTITY_ID = struct<decs::EntityId>`

Entity ID which represents invalid entity.

8.1.3 Structures

`decs::CTypeInfo`

Type information for the individual component subtype. Consists of type name and collection of type-specific routines to control type values during its lifetime, serialization, etc.

Fields

- **basicType** : *Type* - basic type of the component
- **mangledName** : string - mangled name of the type
- **fullName** : string - full name of the type
- **hash** : *TypeHash* - hash of the type
- **size** : uint - size of the type
- **eraser** : function<(arr:array<uint8>):void> - function to erase component value
- **clonner** : function<(dst:array<uint8>;src:array<uint8>):void> - function to clone component value
- **serializer** : function<(arch: *Archive*;arr:array<uint8>;name:string):void> - function to serialize component value
- **dumper** : function<(elem:void?):string> - function to dump component value as text
- **mkTypeInfo** : function<void> - function to make TypeInfo for the component type
- **gc** : function<(src:array<uint8>):lambda<void>> - function to perform GC marking on the component value

`decs::Component`

Single ECS component. Contains component name, data, and data layout.

Fields

- **name** : string - name of the component
- **hash** : *ComponentHash* - hash of the component
- **stride** : int - stride of the component data
- **data** : array<uint8> - raw data of the component
- **info** : *CTypeInfo* - type information of the component
- **gc_dummy** : lambda<void> - this is here so that GC can find real representation of data

`decs::EntityId`

Fields

- **id** : uint - Unique identifier of the entity. Consists of id (index in the data array) and generation.
- **generation** : int - index of the entity

`decs::Archetype`

ECS archetype. Archetype is unique combination of components.

Fields

- **hash** : *ComponentHash* - hash of the archetype (combination of component hashes)
- **components** : array< *Component*> - list of components in the archetype
- **size** : int - number of entities in the archetype
- **eidIndex** : int - index of the 'eid' component in the components array

`decs::ComponentValue`

Value of the component during creation or transformation.

Fields

- **name** : string - name of the component
- **info** : *CTypeInfo* - type information of the component
- **data** : float4[4] - raw data of the component

`decs::EcsRequestPos`

Location of the ECS request in the code (source file and line number).

Fields

- **file** : string - source file
- **line** : uint - line number

`decs::EcsRequest`

Individual ECS requests. Contains list of required components, list of components which are required to be absent. Caches list of archetypes, which match the request.

Fields

- **hash** : *ComponentHash* - hash of the request
- **req** : array<string> - required components
- **reqn** : array<string> - required components which are not present
- **archetypes** : array<int> - sorted list of matching archetypes
- **at** : *EcsRequestPos* - location of the request in the code

`decs::DecsState`

Entire state of the ECS system. Contains archetypes, entities and entity free-list, entity lookup table, all archetypes and archetype lookups, etc.

Fields

- **archetypeLookup** : table< *ComponentHash*;int> - lookup of archetype by its hash
- **allArchetypes** : array< *Archetype*> - all archetypes in the system
- **entityFreeList** : array< *EntityId*> - list of free entity IDs
- **entityLookup** : array<tuple<generation:int;archetype: *ComponentHash*;index:int>> - lookup of entity by its ID

- **componentTypeCheck** : table<string; *CTypeInfo*> - lookup of component type info by its name
- **ecsQueries** : array< *EcsRequest*> - all ECS requests
- **queryLookup** : table< *ComponentHash*;int> - lookup of ECS request by its hash

`decs::DecsPass`

Individual pass of the update of the ECS system. Contains pass name and list of all pass callbacks.

Fields

- **name** : string - name of the pass
- **calls** : array< *PassFunction*> - list of all pass callbacks

8.1.4 Comparison and access

- *ComponentMap. (var cmp: ComponentMap; name: string) : ComponentValue&*
- *EntityId!= (a: EntityId; b: EntityId) : bool*
- *EntityId== (a: EntityId; b: EntityId) : bool*

`ComponentMap.(cmp: ComponentMap; name: string) : ComponentValue&()`

Access to component value by name. For example:

```
create_entity <| @ ( eid, cmp )
    cmp.pos := float3(i)    // same as cmp |> set("pos",float3(i))
```

Arguments

- **cmp** : *ComponentMap*
- **name** : string

`decs::EntityId!=(a: EntityId; b: EntityId) : bool()`

Inequality operator for entity IDs.

Arguments

- **a** : *EntityId* implicit
- **b** : *EntityId* implicit

`decs::EntityId==(a: EntityId; b: EntityId) : bool()`

Equality operator for entity IDs.

Arguments

- **a** : *EntityId* implicit
- **b** : *EntityId* implicit

8.1.5 Access (get/set/clone)

- *clone (var cv: ComponentValue; val: bool)*
- *clone (var cv: ComponentValue; val: EntityId)*
- *clone (var cv: ComponentValue; val: urange)*
- *clone (var cv: ComponentValue; val: range)*
- *clone (var cv: ComponentValue; val: string)*
- *clone (var cv: ComponentValue; val: urange64)*
- *clone (var cv: ComponentValue; val: int)*
- *clone (var cv: ComponentValue; val: range64)*
- *clone (var cv: ComponentValue; val: int64)*
- *clone (var cv: ComponentValue; val: int16)*
- *clone (var cv: ComponentValue; val: int2)*
- *clone (var cv: ComponentValue; val: int3)*
- *clone (var cv: ComponentValue; val: int4)*
- *clone (var cv: ComponentValue; val: int8)*
- *clone (var cv: ComponentValue; val: uint16)*
- *clone (var cv: ComponentValue; val: uint8)*
- *clone (var cv: ComponentValue; val: uint64)*
- *clone (var cv: ComponentValue; val: uint2)*
- *clone (var cv: ComponentValue; val: uint3)*
- *clone (var cv: ComponentValue; val: float)*
- *clone (var cv: ComponentValue; val: uint4)*
- *clone (var cv: ComponentValue; val: float2)*
- *clone (var cv: ComponentValue; val: float3)*
- *clone (var cv: ComponentValue; val: uint)*
- *clone (var cv: ComponentValue; val: float3x4)*
- *clone (var cv: ComponentValue; val: float3x3)*
- *clone (var cv: ComponentValue; val: float4x4)*
- *clone (var cv: ComponentValue; val: double)*
- *clone (var dst: Component; src: Component)*
- *clone (var cv: ComponentValue; val: float4)*
- *get (arch: Archetype; name: string; value: auto(TT)) : auto*
- *get (var cmp: ComponentMap; name: string; var value: auto(TT)) : auto*
- *get_component (eid: EntityId; name: string; defval: auto(TT)) : TT*
- *has (var cmp: ComponentMap; name: string) : bool*
- *has (arch: Archetype; name: string) : bool*

- *remove (var cmp: ComponentMap; name: string)*
- *set (var cmp: ComponentMap; name: string; value: auto(TT)) : auto*
- *set (var cv: ComponentValue; val: auto) : auto*

clone

`decs::`**`clone`**(*cv: ComponentValue; val: bool*)

Sets individual component value. Verifies that the value is of the correct type.

Arguments

- **cv** : *ComponentValue*
- **val** : bool

`decs::`**`clone`**(*cv: ComponentValue; val: EntityId*)

`decs::`**`clone`**(*cv: ComponentValue; val: urange*)

`decs::`**`clone`**(*cv: ComponentValue; val: range*)

`decs::`**`clone`**(*cv: ComponentValue; val: string*)

`decs::`**`clone`**(*cv: ComponentValue; val: urange64*)

`decs::`**`clone`**(*cv: ComponentValue; val: int*)

`decs::`**`clone`**(*cv: ComponentValue; val: range64*)

`decs::`**`clone`**(*cv: ComponentValue; val: int64*)

`decs::`**`clone`**(*cv: ComponentValue; val: int16*)

`decs::`**`clone`**(*cv: ComponentValue; val: int2*)

`decs::`**`clone`**(*cv: ComponentValue; val: int3*)

`decs::`**`clone`**(*cv: ComponentValue; val: int4*)

`decs::`**`clone`**(*cv: ComponentValue; val: int8*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint16*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint8*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint64*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint2*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint3*)

`decs::`**`clone`**(*cv: ComponentValue; val: float*)

`decs::`**`clone`**(*cv: ComponentValue; val: uint4*)

`decs::`**`clone`**(*cv: ComponentValue; val: float2*)

`decs::`**`clone`**(*cv: ComponentValue; val: float3*) decs::**clone**(*cv: ComponentValue; val: uint*)

decs::**clone**(*cv: ComponentValue; val: float3x4*)

decs::**clone**(*cv: ComponentValue; val: float3x3*)

decs::**clone**(*cv: ComponentValue; val: float4x4*)

decs::**clone**(*cv: ComponentValue; val: double*)

decs::**clone**(*dst: Component; src: Component*)

decs::**clone**(*cv: ComponentValue; val: float4*)

---

get

decs::**get(arch: Archetype; name: string; value: auto(TT)) : auto**()

Creates temporary array of component given specific name and type of component. If component is not found - panic.

Arguments

- **arch** : *Archetype*
- **name** : string
- **value** : auto(TT)

decs::**get(cmp: ComponentMap; name: string; value: auto(TT)) : auto**()

---

decs::**get_component(eid: EntityId; name: string; defval: auto(TT)) : TT**()

Returns a copy of the named component for the given entity. If the entity is dead or the component is not found, returns `defval`. The type of the component is inferred from the type of `defval`. Panics if the component exists but its type does not match.

Arguments

- **eid** : *EntityId*
- **name** : string
- **defval** : auto(TT)

has

decs::**has(cmp: ComponentMap; name: string) : bool**()

Returns true if component map has specified component.

Arguments

- **cmp** : *ComponentMap*
- **name** : string

decs::**has(arch: Archetype; name: string) : bool**()

---

decs::**remove**(*cmp: ComponentMap; name: string*)

Removes specified value from the component map.

Arguments

- **cmp** : *ComponentMap*
- **name** : string

set

decs::**set(cmp: ComponentMap; name: string; value: auto(TT)) : auto**()

Set component value specified by name and type. If value already exists, it is overwritten. If already existing value type is not the same - panic.

Arguments

- **cmp** : *ComponentMap*
- **name** : string
- **value** : auto(TT)

decs::**set(cv: ComponentValue; val: auto) : auto**()

8.1.6 Entity status

- *entity_count () : int*
- *is_alive (eid: EntityId) : bool*

decs::**entity_count() : int**()

Returns the total number of alive entities across all archetypes.

decs::**is_alive(eid: EntityId) : bool**()

Returns true if the entity is alive (exists and has not been deleted). An entity is alive when its id is within bounds and its generation matches the lookup table.

Arguments

- **eid** : *EntityId*

8.1.7 Debug and serialization

- *debug_dump ()*
- *describe (info: CTypeInfo) : string*
- *finalize (var cmp: Component)*
- *serialize (var arch: Archive; var src: Component)*

`decs::debug_dump()`

Prints out state of the ECS system.

`decs::describe(info: CTypeInfo) : string()`

Returns textual description of the type.

Arguments

- **info** : *CTypeInfo*

`decs::finalize(`*cmp: Component*`)`

Deletes component.

Arguments

- **cmp** : *Component*

`decs::serialize(`*arch: Archive; src: Component*`)`

Serializes component value.

Arguments

- **arch** : *Archive*
- **src** : *Component*

8.1.8 Stages

- *commit ()*
- *decs_stage (name: string)*
- *register_decs_stage_call (name: string; pcall: PassFunction)*

`decs::commit()`

Finishes all deferred actions.

`decs::decs_stage(`*name: string*`)`

Invokes specific ECS pass. *commit* is called before and after the invocation.

Arguments

- **name** : string

`decs::register_decs_stage_call(`*name: string; pcall: PassFunction*`)`

Registration of a single pass callback. This is a low-level function, used by decs_boost macros.

Arguments

- **name** : string
- **pcall** : *PassFunction*

8.1.9 Deferred actions

- *create_entity (var blk: lambda<(eid:EntityId;var cmp:ComponentMap):void>) : EntityId*
- *delete_entity (entityid: EntityId)*
- *update_entity (entityid: EntityId; var blk: lambda<(eid:EntityId;var cmp:ComponentMap):void>)*

decs::**create_entity(blk: lambda<(eid:EntityId;var cmp:ComponentMap):void>) : EntityId**()

Creates deferred action to create entity.

Arguments

- **blk** : lambda<(eid: *EntityId*;cmp: *ComponentMap*):void>

decs::**delete_entity**(*entityid: EntityId*)

Creates deferred action to delete entity specified by id.

Arguments

- **entityid** : *EntityId* implicit

decs::**update_entity**(*entityid: EntityId; blk: lambda<(eid:EntityId;var cmp:ComponentMap):void>*)

Creates deferred action to update entity specified by id.

Arguments

- **entityid** : *EntityId* implicit
- **blk** : lambda<(eid: *EntityId*;cmp: *ComponentMap*):void>

8.1.10 GC and reset

- *after_gc ()*
- *before_gc ()*
- *restart ()*

decs::**after_gc**()

Low level callback to be called after the garbage collection. This is a low-level function typically used by *live*.

decs::**before_gc**()

Low level callback to be called before the garbage collection. This is a low-level function typically used by *live*.

decs::**restart**()

Restarts ECS by erasing all deferred actions and entire state.

8.1.11 Iteration

- *decs_array (atype: auto(TT); src: array<uint8>; capacity: int) : auto*
- *for_each_archetype (hash: ComponentHash; var erq: function<():void>; blk: block<(arch:Archetype):void>)*
- *for_each_archetype (var erq: EcsRequest; blk: block<(arch:Archetype):void>)*
- *for_each_archetype_find (hash: ComponentHash; var erq: function<():void>; blk: block<(arch:Archetype):bool>) : bool*
- *for_eid_archetype (eid: EntityId; hash: ComponentHash; var erq: function<():void>; blk: block<(arch:Archetype;index:int):void>) : bool*
- *get_default_ro (arch: Archetype; name: string; value: auto(TT)) : iterator<TT const&>*
- *get_optional (arch: Archetype; name: string; value: auto(TT)?) : iterator<TT?>*
- *get_ro (arch: Archetype; name: string; value: auto(TT)) : array<TT>*
- *get_ro (arch: Archetype; name: string; value: auto(TT)[]) : array<TT[-2]>*

decs::**decs_array(atype: auto(TT); src: array<uint8>; capacity: int) : auto**()

Warning: This is unsafe operation.

Low level function returns temporary array of component given specific type of component.

Arguments

- **atype** : auto(TT)
- **src** : array<uint8>
- **capacity** : int

for_each_archetype

decs::**for_each_archetype**(*hash: ComponentHash; erq: function<():void>; blk: block<(arch:Archetype):void>*)

Invokes block for each entity of each archetype that can be processed by the request. Request is returned by a specified function.

Arguments

- **hash** : *ComponentHash*
- **erq** : function<void>
- **blk** : block<(arch: *Archetype*):void>

decs::**for_each_archetype**(*erq: EcsRequest; blk: block<(arch:Archetype):void>*)

---

decs::**for_each_archetype_find(hash: ComponentHash; erq: function<():void>; blk: block<(arch:Archetype):**

Invokes block for each entity of each archetype that can be processed by the request. Request is returned by a specified function. If block returns true, iteration is stopped.

Arguments

- **hash** : *ComponentHash*
- **erq** : function<void>
- **blk** : block<(arch: *Archetype*):bool>

`decs::for_eid_archetype(eid: EntityId; hash: ComponentHash; erq: function<():void>; blk: block<(arch:Ar`

Invokes block for the specific entity id, given request. Request is returned by a specified function.

Arguments

- **eid** : *EntityId* implicit
- **hash** : *ComponentHash*
- **erq** : function<void>
- **blk** : block<(arch: *Archetype*;index:int):void>

`decs::get_default_ro(arch: Archetype; name: string; value: auto(TT)) : iterator<TT const&>()`

Returns const iterator of component given specific name and type of component. If component is not found - iterator will keep returning the specified value.

Arguments

- **arch** : *Archetype*
- **name** : string
- **value** : auto(TT)

`decs::get_optional(arch: Archetype; name: string; value: auto(TT)?) : iterator<TT?>()`

Returns const iterator of component given specific name and type of component. If component is not found - iterator will keep returning default value for the component type.

Arguments

- **arch** : *Archetype*
- **name** : string
- **value** : auto(TT)?

get_ro

`decs::get_ro(arch: Archetype; name: string; value: auto(TT)) : array<TT>()`

Returns const temporary array of component given specific name and type of component for regular components.

Arguments

- **arch** : *Archetype*
- **name** : string
- **value** : auto(TT)

`decs::get_ro(arch: Archetype; name: string; value: auto(TT)[]) : array<TT[-2]>()`

8.1.12 Request

- *EcsRequestPos (at: LineInfo) : EcsRequestPos*
- *compile_request (var erq: EcsRequest)*
- *lookup_request (var erq: EcsRequest) : int*
- *verify_request (var erq: EcsRequest) : tuple<ok:bool;error:string>*

`decs::EcsRequestPos(at: LineInfo) : EcsRequestPos()`

Constructs EcsRequestPos from rtti::LineInfo.

Arguments

- **at** : *LineInfo*

`decs::compile_request(erq: EcsRequest)`

Compiles ECS request, by creating request hash.

Arguments

- **erq** : *EcsRequest*

`decs::lookup_request(erq: EcsRequest) : int()`

Looks up ECS request in the request cache.

Arguments

- **erq** : *EcsRequest*

`decs::verify_request(erq: EcsRequest) : tuple<ok:bool;error:string>()`

Verifies ECS request. Returns pair of boolean (true for OK) and error message.

Arguments

- **erq** : *EcsRequest*

8.2 Boost package for DECS

The DECS_BOOST module provides convenience macros and syntactic sugar for the DECS entity component system, including simplified component registration, entity creation, and system definition patterns.

See also *DECS, Daslang entity component system* for the core ECS runtime and *DECS debug state reporting* for entity state machines. See tutorial_decs for a hands-on tutorial.

All functions and symbols are in "decs_boost" module, use require to get access to it.

```
require daslib/decs_boost
```

Example:

```
options persistent_heap = true
    require daslib/decs_boost

    [export]
    def main() {
```

(continues on next page)

(continued from previous page)

```
        restart()
        create_entity() @(eid, cmp) {
            cmp |> set("pos", float3(1, 2, 3))
            cmp |> set("name", "hero")
        }
        commit()
        query() $(pos : float3; name : string) {
            print("{name} at {pos}\n")
        }
    }
    // output:
    // hero at 1,2,3
```

8.2.1 Function annotations

decs_boost::**REQUIRE**

This annotation provides list of required components for entity.

decs_boost::**REQUIRE_NOT**

This annotation provides list of components, which are required to not be part of the entity.

decs_boost::**decs**

This macro converts a function into a DECS pass stage query. Possible arguments are *stage*, *REQUIRE*, and *REQUIRE_NOT*. It has all other properties of a *query* (like ability to operate on templates). For example:

```
[decs(stage=update_ai, REQUIRE=ai_turret)]
    def update_ai ( eid:EntityId; var turret:Turret; pos:float3 )
        ...
```

In the example above a query is added to the *update_ai* stage. The query also requires that each entity passed to it has an *ai_turret* property.

8.2.2 Call macros

decs_boost::**query**

This macro implements *query* functionality. There are 2 types of queries:

- query(…) - returns a list of entities matching the query
- query(eid) - returns a single entity matching the eid

For example:

```
query() <| $ ( eid:EntityId; pos, vel : float3 )
    print("[{eid}] pos={pos} vel={vel}\n")
```

The query above will print all entities with position and velocity. Here is another example:

```
query(kaboom) <| $ ( var pos:float3&; vel:float3; col:uint=13u )
    pos += vel
```

The query above will add the velocity to the position of an entity with eid kaboom.

Query can have *REQUIRE* and *REQUIRE_NOT* clauses:

```
var average : float3
query <| $ [REQUIRE(tank)] ( pos:float3 )
    average += pos
```

The query above will add *pos* components of all entities, which also have a *tank* component.

Additionally queries can automatically expand components of entities. For example:

```
[decs_template(prefix="particle")]
struct Particle
    pos, vel : float3
...
query <| $ ( var q : Particle )
    q.pos += q.vel                  // this is actually particlepos += particlevel
```

In the example above structure q : Particle does not exist as a variable. Instead it is expanded into accessing individual components of the entity. REQURE section of the query is automatically filled with all components of the template. If template prefix is not specified, prefix is taken from the name of the template (would be `Particle_`). Specifying empty prefix *[decs_template(prefix)]* will result in no prefix being added.

Note: apart from tagging structure as a template, the macro also generates *apply_decs_template* and *remove_decs_template* functions. *apply_decs_template* is used to add template to an entity, and *remove_decs_template* is used to remove all components of the template from the entity:

```
for i in range(3)
    create_entity <| @ ( eid, cmp )
        apply_decs_template(cmp, [[Particle pos=float3(i), vel=float3(i+1)]])
```

`decs_boost::`**`find_query`**

This macro implements *find_query* functionality. It is similar to *query* in most ways, with the main differences being:

- there is no eid-based find query
- the find_query stops once the first match is found

For example:

```
let found = find_query <| $ ( pos,dim:float3; obstacle:Obstacle )
if !obstacle.wall
    return false
let aabb = [[AABB min=pos-dim*0.5, max=pos+dim*0.5 ]]
if is_intersecting(ray, aabb, 0.1, dist)
    return true
```

In the example above the find_query will return *true* once the first intersection is found. Note: if return is missing, or end of find_query block is reached - its assumed that find_query did not find anything, and will return false.

`decs_boost::`**`from_decs`**

This macro converts a DECS query into an iterator<tuple<…>>. For example:

```
let it = from_decs($(index:int; text:string){})
for (item in it) {
```

(continues on next page)

(continued from previous page)

```
    // process item
    print("Entity {item.index}: {item.text}\n")
}
```

Internally it generates the following code:

```
let it = invoke($() {
    var res : array<tuple<int,string>>
    query($(index:int; text:string) {
        res |> push((index, text))
    })
    return res.to_sequence()
```

8.2.3 Structure macros

decs_boost::**decs_template**

This macro creates a template for the given structure. *apply_decs_template* and *remove_decs_template* functions are generated for the structure type.

8.3 DECS debug state reporting

The DECS_STATE module extends DECS with state machine support for entities. It provides state transition management, allowing entities to change behavior based on their current state.

See also *DECS, Daslang entity component system* for the core ECS runtime and *Boost package for DECS* for query macros. See tutorial_decs for a hands-on tutorial.

All functions and symbols are in "decs_state" module, use require to get access to it.

```
require daslib/decs_state
```

CHAPTER NINE

CONCURRENCY

Job queues, coroutines, asynchronous operations, and cross-context invocation.

9.1 Jobs and threads

The JOBQUE module provides low-level job queue and threading primitives. It includes thread-safe channels for inter-thread communication, lock boxes for shared data access, job status tracking, and fine-grained thread management. For higher-level job abstractions, see `jobque_boost`.

See tutorial_jobque for a hands-on tutorial.

All functions and symbols are in “jobque” module, use require to get access to it.

```
require jobque
```

Example:

```
require jobque

    [export]
    def main() {
        with_atomic32() $(counter) {
            counter |> set(10)
            print("value = {counter |> get}\n")
            let after_inc = counter |> inc
            print("after inc = {after_inc}\n")
            let after_dec = counter |> dec
            print("after dec = {after_dec}\n")
        }
    }
    // output:
    // value = 10
    // after inc = 11
    // after dec = 10
```

9.1.1 Handled structures

`jobque::JobStatus`

`JobStatus.isReady() : bool()`

Whether the job has completed execution.

`JobStatus.isValid() : bool()`

Whether the job status object refers to a valid, active job.

`JobStatus.size() : int()`

Returns the current entry count of the JobStatus or Channel.

Properties

- **isReady** : bool
  - **isValid** : bool
  - **size** : int

Job status indicator (ready or not, as well as entry count).

`jobque::Channel`

`Channel.isEmpty() : bool()`

Whether the channel or pipe contains no remaining elements.

`Channel.total() : int()`

Total number of elements that have been added to the pipe.

Properties

- **isEmpty** : bool
  - **total** : int

Channel provides a way to communicate between multiple contexts, including threads and jobs. Channel has internal entry count.

`jobque::LockBox`

Lockbox. Similar to channel, only for single object.

`jobque::Atomic32`

Atomic 32 bit integer.

`jobque::Atomic64`

Atomic 64 bit integer.

9.1.2 Channel, JobStatus, Lockbox

- *add_ref (status: JobStatus?)*
- *append (channel: JobStatus?; size: int) : int*
- *channel_create () : Channel?*
- *channel_remove (channel: Channel?&)*
- *job_status_create () : JobStatus?*
- *job_status_remove (jobStatus: JobStatus?&)*
- *join (job: JobStatus?)*
- *lock_box_create () : LockBox?*
- *lock_box_remove (box: LockBox?&)*
- *notify (job: JobStatus?)*
- *notify_and_release (job: JobStatus?&)*
- *release (status: JobStatus?&)*

`jobque::`**`add_ref`**(*status: JobStatus?*)

Increases the reference count of a `JobStatus` or `Channel`, preventing premature deletion.

Arguments

- **status** : *JobStatus*? implicit

`jobque::`**`append(channel: JobStatus?; size: int) : int`**()

Increases the entry count of the channel, signaling that new work has been added.

Arguments

- **channel** : *JobStatus*? implicit
- **size** : int

`jobque::`**`channel_create() : Channel?`**()

Warning: This is unsafe operation.

Creates a new `Channel` for inter-thread communication and synchronization.

`jobque::`**`channel_remove`**(*channel: Channel?&*)

Warning: This is unsafe operation.

Destroys a `Channel` and releases its resources.

Arguments

- **channel** : *Channel*?& implicit

jobque::**job_status_create() : JobStatus?**()

Creates a new `JobStatus` object for tracking the completion state of asynchronous jobs.

jobque::**job_status_remove**(*jobStatus: JobStatus?&*)

Warning: This is unsafe operation.

Destroys a `JobStatus` object and releases its resources.

Arguments

- **jobStatus** : *JobStatus*?& implicit

jobque::**join**(*job: JobStatus?*)

Blocks the current thread until the job or channel's entry count reaches zero, indicating all work is complete.

Arguments

- **job** : *JobStatus*? implicit

jobque::**lock_box_create() : LockBox?**()

Creates a new `LockBox` for thread-safe shared access to a single value.

jobque::**lock_box_remove**(*box: LockBox?&*)

Warning: This is unsafe operation.

Destroys a `LockBox` and releases its resources.

Arguments

- **box** : *LockBox*?& implicit

jobque::**notify**(*job: JobStatus?*)

Decreases the channel's entry count, signaling that one unit of work has completed.

Use `notify` when the caller does **not** own a reference to the channel — for example when a `Channel?` is passed as a plain function argument via `invoke_in_context`. In that scenario no lambda captures the channel, so no extra reference was added and there is nothing to release.

Compare with `notify_and_release`, which additionally releases a reference and should be used inside lambdas that captured the channel (adding a reference).

Arguments

- **job** : *JobStatus*? implicit

jobque::**notify_and_release**(*job: JobStatus?&*)

Decreases the entry count **and** the reference count of a `Channel` or `JobStatus` in a single operation. After the call the channel/status variable is set to `null`.

Use `notify_and_release` inside lambdas that captured the channel. Capturing adds a reference, so the lambda must release it when done. This function combines `notify` + `release` into one atomic step and nulls the variable to prevent accidental reuse.

If the caller does **not** own a reference (e.g. the channel was passed as a plain argument via `invoke_in_context`, with no lambda capture), use `notify` instead — calling `notify_and_release` in that case would release a reference the caller never added, leading to a premature free.

Arguments

- **job** : *JobStatus*?& implicit

`jobque::`**`release`**(*status: JobStatus?&*)

Decreases the reference count of a `JobStatus` or `Channel`; the object is deleted when the count reaches zero.

Arguments

- **status** : *JobStatus*?& implicit

9.1.3 Queries

- *get_total_hw_jobs () : int*
- *get_total_hw_threads () : int*
- *is_job_que_shutting_down () : bool*

`jobque::`**`get_total_hw_jobs() : int`**`()`

Returns the total number of hardware threads allocated to the job system.

`jobque::`**`get_total_hw_threads() : int`**`()`

Returns the total number of hardware threads available on the system.

`jobque::`**`is_job_que_shutting_down() : bool`**`()`

Returns `true` if the job queue infrastructure is shutting down or has not been initialized.

9.1.4 Internal invocations

- *new_debugger_thread (block: block<():void>)*
- *new_job_invoke (lambda: lambda<():void>; function: function<():void>; lambdaSize: int)*
- *new_thread_invoke (lambda: lambda<():void>; function: function<():void>; lambdaSize: int)*

`jobque::`**`new_debugger_thread`**(*block: block<():void>*)

Creates a new debugger tick thread for servicing debug connections.

Arguments

- **block** : block<void> implicit

`jobque::`**`new_job_invoke`**(*lambda: lambda<():void>; function: function<():void>; lambdaSize: int*)

Clones the current context, moves the attached lambda into it, and submits it to the job queue.

Arguments

- **lambda** : lambda<void>
- **function** : function<void>
- **lambdaSize** : int

`jobque::`**`new_thread_invoke`**(*lambda: lambda<():void>; function: function<():void>; lambdaSize: int*)

Clones the current context, moves the attached lambda into it, and runs it on a new dedicated thread.

Arguments

- **lambda** : lambda<void>
- **function** : function<void>
- **lambdaSize** : int

9.1.5 Construction

- *with_channel (block: block<(Channel?):void>)*
- *with_channel (count: int; block: block<(Channel?):void>)*
- *with_job_que (block: block<():void>)*
- *with_job_status (total: int; block: block<(JobStatus?):void>)*
- *with_lock_box (block: block<(LockBox?):void>)*

with_channel

`jobque::`**`with_channel`**(*block: block<(Channel?):void>*)

Creates a `Channel` scoped to the given block and automatically destroys it afterward.

Arguments

- **block** : block<(*Channel*?):void> implicit

`jobque::`**`with_channel`**(*count: int; block: block<(Channel?):void>*)

`jobque::`**`with_job_que`**(*block: block<():void>*)

Ensures job queue infrastructure is initialized for the duration of the block.

Arguments

- **block** : block<void> implicit

`jobque::`**`with_job_status`**(*total: int; block: block<(JobStatus?):void>*)

Creates a `JobStatus` scoped to the given block and automatically destroys it afterward.

Arguments

- **total** : int
- **block** : block<(*JobStatus*?):void> implicit

`jobque::`**`with_lock_box`**(*block: block<(LockBox?):void>*)

Creates a `LockBox` scoped to the given block and automatically destroys it afterward.

Arguments

- **block** : block<(*LockBox*?):void> implicit

9.1.6 Atomic

- *atomic32_create () : Atomic32?*
- *atomic32_remove (atomic: Atomic32?&)*
- *atomic64_create () : Atomic64?*
- *atomic64_remove (atomic: Atomic64?&)*
- *dec (atomic: Atomic32?) : int*
- *dec (atomic: Atomic64?) : int64*
- *get (atomic: Atomic32?) : int*
- *get (atomic: Atomic64?) : int64*
- *inc (atomic: Atomic32?) : int*
- *inc (atomic: Atomic64?) : int64*
- *set (atomic: Atomic32?; value: int)*
- *set (atomic: Atomic64?; value: int64)*
- *with_atomic32 (block: block<(Atomic32?):void>)*
- *with_atomic64 (block: block<(Atomic64?):void>)*

`jobque::`**`atomic32_create() : Atomic32?`**`()`

Creates an `Atomic32` — a thread-safe 32-bit integer for lock-free concurrent access.

`jobque::`**`atomic32_remove`**`(`*atomic: Atomic32?&*`)`

Warning: This is unsafe operation.

Destroys an `Atomic32` and releases its resources.

Arguments

- **atomic** : *Atomic32*?& implicit

`jobque::`**`atomic64_create() : Atomic64?`**`()`

Creates an `Atomic64` — a thread-safe 64-bit integer for lock-free concurrent access.

`jobque::`**`atomic64_remove`**`(`*atomic: Atomic64?&*`)`

Warning: This is unsafe operation.

Destroys an `Atomic64` and releases its resources.

Arguments

- **atomic** : *Atomic64*?& implicit

dec

`jobque::dec(atomic: Atomic32?) : int()`

Atomically decrements the integer value and returns the result.

Arguments

- **atomic** : *Atomic32*? implicit

`jobque::dec(atomic: Atomic64?) : int64()`

---

get

`jobque::get(atomic: Atomic32?) : int()`

Returns the current value of the atomic integer.

Arguments

- **atomic** : *Atomic32*? implicit

`jobque::get(atomic: Atomic64?) : int64()`

---

inc

`jobque::inc(atomic: Atomic32?) : int()`

Atomically increments the integer value and returns the result.

Arguments

- **atomic** : *Atomic32*? implicit

`jobque::inc(atomic: Atomic64?) : int64()`

---

set

`jobque::set`(*atomic: Atomic32?; value: int*)

Sets the atomic integer to the specified value.

Arguments

- **atomic** : *Atomic32*? implicit
- **value** : int

`jobque::set`(*atomic: Atomic64?; value: int64*)

---

jobque::**with_atomic32**(*block: block<(Atomic32?):void>*)

Creates an `Atomic32` scoped to the given block and automatically destroys it afterward.

Arguments

- **block** : block<(*Atomic32*?):void> implicit

jobque::**with_atomic64**(*block: block<(Atomic64?):void>*)

Creates an `Atomic64` scoped to the given block and automatically destroys it afterward.

Arguments

- **block** : block<(*Atomic64*?):void> implicit

9.2 Boost package for jobs and threads

The JOBQUE_BOOST module provides high-level job queue abstractions built on the low-level `jobque` primitives. It includes `with_job`, `with_job_status`, and channel-based patterns for simplified concurrent programming.

See also *Jobs and threads* for the low-level job queue primitives. See tutorial_jobque for a hands-on tutorial.

All functions and symbols are in "jobque_boost" module, use require to get access to it.

```
require daslib/jobque_boost
```

Example:

```
require daslib/jobque_boost

    [export]
    def main() {
        with_job_status(1) $(status) {
            new_thread() @() {
                print("from thread\n")
                status |> notify_and_release()
            }
            status |> join()
            print("thread done\n")
        }
    }
    // output:
    // from thread
    // thread done
```

9.2.1 Function annotations

jobque_boost::**NewJobMacro**

this macro handles *new_job* and *new_thread* calls. the call is replaced with *new_job_invoke* and *new_thread_invoke* accordingly. a cloning infrastructure is generated for the lambda, which is invoked in the new context.

jobque_boost::**ParallelForJobMacro**

Base macro for parallel_for, parallel_for_each, and parallel_map. Wraps the block body in `new_job` and redirects to the runtime implementation.

jobque_boost::**ParallelForEachJobMacro**

This macro handles *parallel_for_each*. Wraps block body in `new_job` and redirects to `_parallel_for_each`.

jobque_boost::**ParallelMapJobMacro**

This macro handles *parallel_map*. Wraps block body in `new_job` and redirects to `_parallel_map`.

9.2.2 Invocations

- *new_job (var l: lambda<():void>)*
- *new_thread (var l: lambda<():void>)*

jobque_boost::**new_job**(*l: lambda<():void>*)

Create a new job.

- new context is cloned from the current context.
- lambda is cloned to the new context.
- new job is added to the job queue.
- once new job is invoked, lambda is invoked on the new context on the job thread.

Arguments

- **l** : lambda<void>

jobque_boost::**new_thread**(*l: lambda<():void>*)

Create a new thread

- new context is cloned from the current context.
- lambda is cloned to the new context.
- new thread is created.
- lambda is invoked on the new context on the new thread.

Arguments

- **l** : lambda<void>

9.2.3 Iteration

- *each (var channel: Channel?; tinfo: auto(TT)) : auto*
- *each_clone (var channel: Channel?; tinfo: auto(TT)) : auto*
- *for_each (channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto*
- *for_each_clone (channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto*

jobque_boost::**each(channel: Channel?; tinfo: auto(TT)) : auto**()

Warning: This function is deprecated.

this iterator is used to iterate over the channel in order it was pushed. iterator stops once channel is depleted (internal entry counter is 0) iteration can happen on multiple threads or jobs at the same time.

Arguments

- **channel** : *Channel*?
- **tinfo** : auto(TT)

jobque_boost::**each_clone(channel: Channel?; tinfo: auto(TT)) : auto**()

this iterator is used to iterate over the channel in order it was pushed. iterator stops once channel is depleted (internal entry counter is 0) iteration can happen on multiple threads or jobs at the same time.

Arguments

- **channel** : *Channel*?
- **tinfo** : auto(TT)

jobque_boost::**for_each(channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto**()

Warning: This function is deprecated.

reads input from the channel (in order it was pushed) and invokes the block on each input. stops once channel is depleted (internal entry counter is 0) this can happen on multiple threads or jobs at the same time.

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

jobque_boost::**for_each_clone(channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto**()

reads input from the channel (in order it was pushed) and invokes the block on each input. stops once channel is depleted (internal entry counter is 0) this can happen on multiple threads or jobs at the same time.

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

9.2.4 Passing data

- *push (channel: Channel?; data: auto?) : auto*
- *push_batch (channel: Channel?; data: array<auto?>) : auto*
- *push_batch_clone (channel: Channel?; data: array<auto(TT)>) : auto*
- *push_clone (channel: Channel?; data: auto(TT)) : auto*

jobque_boost::**push(channel: Channel?; data: auto?) : auto**()

pushes value to the channel (at the end)

Arguments

- **channel** : *Channel*?
- **data** : auto?

jobque_boost::**push_batch(channel: Channel?; data: array<auto?>) : auto**()

pushes values to the channel (at the end)

Arguments

- **channel** : *Channel*?
- **data** : array<auto?>

jobque_boost::**push_batch_clone(channel: Channel?; data: array<auto(TT)>) : auto**()

clones data and pushes values to the channel (at the end)

Arguments

- **channel** : *Channel*?
- **data** : array<auto(TT)>

jobque_boost::**push_clone(channel: Channel?; data: auto(TT)) : auto**()

clones data and pushes value to the channel (at the end)

Arguments

- **channel** : *Channel*?
- **data** : auto(TT)

9.2.5 Receiving data

- *gather (ch: Channel?; blk: block<(arg:auto(TT)#):void>) : auto*
- *gather_and_forward (ch: Channel?; toCh: Channel?; blk: block<(arg:auto(TT)#):void>) : auto*
- *gather_ex (ch: Channel?; blk: block<(arg:auto(TT)#;info:TypeInfo const?;var ctx:Context):void>) : auto*
- *peek (ch: Channel?; blk: block<(arg:auto(TT)#):void>) : auto*
- *pop_and_clone_one (channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto*
- *pop_one (channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto*
- *pop_with_timeout (channel: Channel?; timeout_ms: int; blk: block<(res:auto(TT)#):void>) : bool*
- *pop_with_timeout_clone (channel: Channel?; timeout_ms: int; blk: block<(res:auto(TT)#):void>) : bool*
- *try_pop (channel: Channel?; blk: block<(res:auto(TT)#):void>) : bool*
- *try_pop_clone (channel: Channel?; blk: block<(res:auto(TT)#):void>) : bool*

jobque_boost::**gather(ch: Channel?; blk: block<(arg:auto(TT)#):void>) : auto**()

reads input from the channel (in order it was pushed) and invokes the block on each input. afterwards input is consumed

Arguments

- **ch** : *Channel*?
- **blk** : block<(arg:auto(TT)#):void>

`jobque_boost::`**`gather_and_forward(ch: Channel?; toCh: Channel?; blk: block<(arg:auto(TT)#):void>) : auto`**

reads input from the channel (in order it was pushed) and invokes the block on each input. afterwards input is consumed

Arguments

- **ch** : *Channel*?
- **toCh** : *Channel*?
- **blk** : block<(arg:auto(TT)#):void>

`jobque_boost::`**`gather_ex(ch: Channel?; blk: block<(arg:auto(TT)#;info:TypeInfo const?;var ctx:Context):v`**

reads input from the channel (in order it was pushed) and invokes the block on each input. afterwards input is consumed

Arguments

- **ch** : *Channel*?
- **blk** : block<(arg:auto(TT)#;info: *TypeInfo*?;ctx: *Context*):void>

`jobque_boost::`**`peek(ch: Channel?; blk: block<(arg:auto(TT)#):void>) : auto`**`()`

reads input from the channel (in order it was pushed) and invokes the block on each input. afterwards input is not consumed

Arguments

- **ch** : *Channel*?
- **blk** : block<(arg:auto(TT)#):void>

`jobque_boost::`**`pop_and_clone_one(channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto`**`()`

reads one command from channel

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

`jobque_boost::`**`pop_one(channel: Channel?; blk: block<(res:auto(TT)#):void>) : auto`**`()`

Warning: This function is deprecated.

reads one command from channel

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

`jobque_boost::`**`pop_with_timeout(channel: Channel?; timeout_ms: int; blk: block<(res:auto(TT)#):void>) :`**

Pop from channel with timeout in milliseconds. Returns true if an item was available within the timeout, false if timed out or channel exhausted.

Arguments

- **channel** : *Channel*?
- **timeout_ms** : int
- **blk** : block<(res:auto(TT)#):void>

`jobque_boost::pop_with_timeout_clone(channel: Channel?; timeout_ms: int; blk: block<(res:auto(TT)#):voi`

Pop from channel with timeout and clone. Returns true if an item was available within the timeout. The popped value is cloned to the current context before invoking the block.

Arguments

- **channel** : *Channel*?
- **timeout_ms** : int
- **blk** : block<(res:auto(TT)#):void>

`jobque_boost::try_pop(channel: Channel?; blk: block<(res:auto(TT)#):void>) : bool()`

Non-blocking pop from channel. Returns true if an item was available, false if channel was empty. Does not wait for data — returns immediately.

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

`jobque_boost::try_pop_clone(channel: Channel?; blk: block<(res:auto(TT)#):void>) : bool()`

Non-blocking pop with clone from channel. Returns true if an item was available, false if channel was empty. The popped value is cloned to the current context before invoking the block.

Arguments

- **channel** : *Channel*?
- **blk** : block<(res:auto(TT)#):void>

9.2.6 Synchronization

- *done (var status: JobStatus?&)*
- *with_wait_group (blk: block<(var status:JobStatus?):void>)*
- *with_wait_group (count: int; blk: block<(var status:JobStatus?):void>)*

`jobque_boost::done(status: JobStatus?&)`

Mark one unit of work as done in a wait group. Alias for `notify_and_release`. Decrements the notification counter and releases the reference. Sets the pointer to null, preventing double-release.

Arguments

- **status** : *JobStatus*?&

with_wait_group

jobque_boost::**with_wait_group**(*blk: block<(var status:JobStatus?):void>*)

Creates a wait group starting at count 0 with auto-join. Use append to dynamically add expected notifications before dispatching work. The block returns only after all notifications have been received.

Arguments

- **blk** : block<(status: *JobStatus*?):void>

jobque_boost::**with_wait_group**(*count: int; blk: block<(var status:JobStatus?):void>*)

9.2.7 Parallel execution

- *_parallel_for (range_begin: int; range_end: int; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>)*
- *_parallel_for_each (arr: array<auto(TT)>; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>) : auto*
- *_parallel_map (arr: array<auto(TT)>; num_jobs: int; var results_channel: Channel?; blk: block<(job_begin:int;job_end:int;var ch:Channel?;var wg:JobStatus?):void>) : auto*
- *parallel_for (range_begin: int; range_end: int; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>)*
- *parallel_for_each (arr: array<auto(TT)>; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>) : auto*
- *parallel_map (arr: array<auto(TT)>; num_jobs: int; var results_channel: Channel?; blk: block<(job_begin:int;job_end:int;var ch:Channel?;var wg:JobStatus?):void>) : auto*

jobque_boost::**_parallel_for**(*range_begin: int; range_end: int; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>*)

Partitions [range_begin..range_end) into num_jobs chunks and invokes blk once per chunk on the calling thread with (chunk_begin, chunk_end, wg). The block is expected to dispatch work via new_job and call wg |> notify_and_release when each job finishes. parallel_for blocks until all notifications are received (via internal with_wait_group). Requires with_job_que context.

Arguments

- **range_begin** : int
- **range_end** : int
- **num_jobs** : int
- **blk** : block<(job_begin:int;job_end:int;wg: *JobStatus*?):void>

jobque_boost::**_parallel_for_each(arr: array<auto(TT)>; num_jobs: int; blk: block<(job_begin:int;job_end**

Runtime implementation for parallel_for_each. Partitions array indices [0..length(arr)) into num_jobs chunks and invokes blk with (chunk_begin_idx, chunk_end_idx, wg) on the calling thread. The block should dispatch new_job calls that process arr[i] for i in [chunk_begin_idx..chunk_end_idx), then call wg |> notify_and_release. Blocks until all jobs finish. Requires with_job_que context.

Arguments

- **arr** : array<auto(TT)>
- **num_jobs** : int

- **blk** : block<(job_begin:int;job_end:int;wg: *JobStatus*?):void>

`jobque_boost::`**`_parallel_map(arr: array<auto(TT)>; num_jobs: int; results_channel: Channel?; blk: block<`**

Runtime implementation for `parallel_map`. Partitions array indices `[0..length(arr))` into `num_jobs` chunks and invokes `blk` on the calling thread with `(chunk_begin_idx, chunk_end_idx, results_channel, wg)`. Blocks until all jobs finish. Results are available in `results_channel` after this call returns. Requires `with_job_que` context.

Arguments

- **arr** : array<auto(TT)>
- **num_jobs** : int
- **results_channel** : *Channel*?
- **blk** : block<(job_begin:int;job_end:int;ch: *Channel*?;wg: *JobStatus*?):void>

`jobque_boost::`**`parallel_for`**(*range_begin: int; range_end: int; num_jobs: int; blk: block<(job_begin:int;job_end:int;var wg:JobStatus?):void>*)

this one is stub for _parallel_for

Arguments

- **range_begin** : int
- **range_end** : int
- **num_jobs** : int
- **blk** : block<(job_begin:int;job_end:int;wg: *JobStatus*?):void>

`jobque_boost::`**`parallel_for_each(arr: array<auto(TT)>; num_jobs: int; blk: block<(job_begin:int;job_end:`**

Convenience wrapper around `parallel_for` for arrays. Partitions array indices `[0..length(arr))` into `num_jobs` chunks. The block body is automatically wrapped in `new_job`. Blocks until all jobs finish. Requires `with_job_que` context.

Arguments

- **arr** : array<auto(TT)>
- **num_jobs** : int
- **blk** : block<(job_begin:int;job_end:int;wg: *JobStatus*?):void>

`jobque_boost::`**`parallel_map(arr: array<auto(TT)>; num_jobs: int; results_channel: Channel?; blk: block<(`**

Partitions array indices `[0..length(arr))` into `num_jobs` chunks. The block body is automatically wrapped in `new_job`. Blocks until all jobs finish. Results are available in `results_channel` after this call returns. Requires `with_job_que` context.

Arguments

- **arr** : array<auto(TT)>
- **num_jobs** : int
- **results_channel** : *Channel*?
- **blk** : block<(job_begin:int;job_end:int;ch: *Channel*?;wg: *JobStatus*?):void>

9.2.8 LockBox operations

- *clear (box: LockBox?; type_: auto(TT)) : auto*
- *get (box: LockBox?; blk: block<(res:auto(TT)#):void>) : auto*
- *set (box: LockBox?; data: auto?) : auto*
- *set (box: LockBox?; data: auto(TT)) : auto*
- *update (box: LockBox?; blk: block<(var res:auto(TT)#):void>) : auto*

`jobque_boost::clear(box: LockBox?; type_: auto(TT)) : auto()`

clear value from the lock box

Arguments

- **box** : *LockBox*?
- **type_** : auto(TT)

`jobque_boost::get(box: LockBox?; blk: block<(res:auto(TT)#):void>) : auto()`

reads value from the lock box and invokes the block on it

Arguments

- **box** : *LockBox*?
- **blk** : block<(res:auto(TT)#):void>

set

`jobque_boost::set(box: LockBox?; data: auto?) : auto()`

sets value to the lock box

Arguments

- **box** : *LockBox*?
- **data** : auto?

`jobque_boost::set(box: LockBox?; data: auto(TT)) : auto()`

`jobque_boost::update(box: LockBox?; blk: block<(var res:auto(TT)#):void>) : auto()`

update value in the lock box and invokes the block on it

Arguments

- **box** : *LockBox*?
- **blk** : block<(res:auto(TT)#):void>

9.2.9 Internal capture details

- *capture_jobque_channel (var ch: Channel?) : Channel?*
- *capture_jobque_job_status (var js: JobStatus?) : JobStatus?*
- *capture_jobque_lock_box (var js: LockBox?) : LockBox?*
- *release_capture_jobque_channel (ch: Channel?)*
- *release_capture_jobque_job_status (js: JobStatus?)*
- *release_capture_jobque_lock_box (js: LockBox?)*

`jobque_boost::`**`capture_jobque_channel(ch: Channel?) : Channel?`**`()`

this function is used to capture a channel that is used by the jobque.

Arguments

- **ch** : *Channel*?

`jobque_boost::`**`capture_jobque_job_status(js: JobStatus?) : JobStatus?`**`()`

this function is used to capture a job status that is used by the jobque.

Arguments

- **js** : *JobStatus*?

`jobque_boost::`**`capture_jobque_lock_box(js: LockBox?) : LockBox?`**`()`

this function is used to capture a lock box that is used by the jobque.

Arguments

- **js** : *LockBox*?

`jobque_boost::`**`release_capture_jobque_channel`**(*ch: Channel?*)

this function is used to release a channel that is used by the jobque.

Arguments

- **ch** : *Channel*?

`jobque_boost::`**`release_capture_jobque_job_status`**(*js: JobStatus?*)

this function is used to release a job status that is used by the jobque.

Arguments

- **js** : *JobStatus*?

`jobque_boost::`**`release_capture_jobque_lock_box`**(*js: LockBox?*)

this function is used to release a lock box that is used by the jobque.

Arguments

- **js** : *LockBox*?

9.3 Cross-context evaluation helpers

The APPLY_IN_CONTEXT module extends apply operations to work across different execution contexts, enabling cross-context function invocation with packed arguments.

See tutorial_apply_in_context for a hands-on tutorial.

All functions and symbols are in "apply_in_context" module, use require to get access to it.

```
require daslib/apply_in_context
```

9.3.1 Function annotations

apply_in_context::**apply_in_context**

[apply_in_context] function annotation. Function is modified, so that it is called in the debug agent context, specified in the annotation. If specified context is not installed, panic is called.

For example:

```
[apply_in_context(opengl_cache)]
def public cache_font(name:string implicit) : Font?
    ...
...
let font = cache_font("Arial") // call invoked in the "opengl_cache" debug agent context
```

9.4 Coroutines and additional generator support

The COROUTINES module provides coroutine infrastructure including the `[coroutine]` function annotation, `yield_from` for delegating to sub-coroutines, and `co_await` for composing asynchronous generators. Coroutines produce values lazily via `yield` and can be iterated with `for`.

See tutorial_iterators_and_generators for a hands-on tutorial.

All functions and symbols are in "coroutines" module, use require to get access to it.

```
require daslib/coroutines
```

Example:

```
require daslib/coroutines

    [coroutine]
    def fibonacci() : int {
        var a = 0
        var b = 1
        while (true) {
            yield a
            let next = a + b
            a = b
            b = next
        }
```

(continues on next page)

(continued from previous page)

```
    }

    [export]
    def main() {
        var count = 0
        for (n in fibonacci()) {
            print("{n} ")
            count ++
            if (count >= 10) {
                break
            }
        }
        print("\n")
    }
    // output:
    // 0 1 1 2 3 5 8 13 21 34
```

9.4.1 Type aliases

coroutines::**Coroutine = iterator<bool>**

A coroutine is a generator that yields bool to indicate if it is still running.

coroutines::**Coroutines = array<iterator<bool>>**

An array of coroutines.

9.4.2 Function annotations

coroutines::**coroutine**

This macro converts coroutine function into generator, adds return false. Daslang implementation of coroutine is generator based. Function is converted into a state machine, which can be resumed and suspended. The function is converted into a generator. Generator yields bool if its a void coroutine, and yields the return type otherwise. If return type is specified coroutine can serve as an advanced form of a generator.

9.4.3 Call macros

coroutines::**co_continue**

This macro converts co_continue to yield true. The idea is that coroutine without specified type is underneath a coroutine which yields bool. That way co_continue() does not distract from the fact that it is a generator<bool>.

coroutines::**co_await**

This macro converts co_await(sub_coroutine) into:

```
for t in subroutine
    yield t
```

The idea is that coroutine or generator can wait for a sub-coroutine to finish.

`coroutines::`**`yeild_from`**

This macro converts yield_from(THAT) expression into:

```
for t in THAT
    yield t
```

The idea is that coroutine or generator can continuously yield from another sub-coroutine or generator.

9.4.4 Top level coroutine evaluation

- *cr_run (var a: Coroutine)*
- *cr_run_all (var a: Coroutines)*

`coroutines::`**`cr_run`**`(`*a: Coroutine*`)`

This function runs coroutine until it is finished.

Arguments

- **a** : *Coroutine*

`coroutines::`**`cr_run_all`**`(`*a: Coroutines*`)`

This function runs all coroutines until they are finished.

Arguments

- **a** : *Coroutines*

9.5 Async/await coroutine macros

The ASYNC_BOOST module implements an async/await pattern for daslang using generator-based cooperative multitasking. It provides the `[async]` function annotation, `await` for waiting on results, and `await_next_frame` for suspending until the next step. Under the hood every `[async]` function is transformed into a state-machine generator — no threads, channels, or job queues are involved.

All functions and symbols are in “async_boost” module, use require to get access to it.

```
require daslib/async_boost
```

See also:

tutorial_async — Tutorial 49: Async / Await.

Coroutines and additional generator support — coroutines module (underlying generator framework).

9.5.1 Function annotations

async_boost::**AwaitMacro**

Function annotation that implements coroutine await semantics.

async_boost::**AwaitCoroutineMacro**

This macro converts await(<coroutine>) expression into:

```
for t in THAT
    yield t
```

The idea is that coroutine or generator can continuously yield from another sub-coroutine or generator.

async_boost::**async**

This macro converts function into generator. Generator yields bool if its a void function (coroutine), and yields the return type otherwise (async return). async function can wait for another async function using await(<async fn call>). use 'return false' to immediately return from the generator.

9.5.2 Awaiting

- *await (var a: iterator<bool>) : bool*
- *await (var a: iterator<variant<res:auto(T);wait:bool>>) : T*
- *await_next_frame ()*

await

async_boost::**await(a: iterator<bool>) : bool**()

This function is used to wait for the result of the async function.

Arguments

- **a** : iterator<bool>

async_boost::**await(a: iterator<variant<res:auto(T);wait:bool>>) : T**()

---

async_boost::**await_next_frame**()

This function is used to suspend coroutine until next frame.

9.5.3 Running async tasks

- *async_run (var a: iterator<auto>) : auto*
- *async_run_all (var a: array<iterator<auto>>) : auto*

async_boost::**async_run(a: iterator<auto>) : auto**()

This function runs async function until it is finished.

Arguments

- **a** : iterator<auto>

async_boost::**async_run_all(a: array<iterator<auto>>) : auto**()

This function runs all async function until they are finished (in parallel, starting from the last one).

Arguments

- **a** : array<iterator<auto>>

9.5.4 Uncategorized

async_boost::**async_timeout(a: iterator<auto>; max_frames: int) : bool**()

This function runs an async function for at most *max_frames* frames. Returns `true` if the async function completed within the limit, `false` if it was terminated due to timeout.

Arguments

- **a** : iterator<auto>
- **max_frames** : int

async_boost::**async_race(a: iterator<auto>; b: iterator<auto>) : int**()

This function runs two async functions concurrently and returns the index (0 or 1) of whichever finishes first. The other is abandoned.

Arguments

- **a** : iterator<auto>
- **b** : iterator<auto>

CHAPTER

TEN

AST AND MACROS

Runtime type information, AST access and manipulation, code generation templates, quoting, and macro infrastructure.

10.1 Runtime type information library

The RTTI module exposes runtime type information and program introspection facilities. It allows querying module structure, type declarations, function signatures, annotations, and other compile-time metadata at runtime. Used primarily by macro libraries and code generation tools.

All functions and symbols are in "rtti" module, use require to get access to it.

```
require rtti
```

10.1.1 Type aliases

rtti::**bitfield ProgramFlags**

Flags which represent state of the *Program* object, both during and after compilation.

Fields

- **failToCompile** (0x1) - indicates that the program failed to compile.
- **_unsafe** (0x2) - indicates that the program contains unsafe code.
- **isCompiling** (0x4) - indicates that the program is currently compiling.
- **isSimulating** (0x8) - indicates that the program is currently simulating.
- **isCompilingMacros** (0x10) - indicates that the program is currently compiling macros.
- **needMacroModule** (0x20) - indicates that the program needs a macro module.
- **promoteToBuiltin** (0x40) - indicates that the program is being promoted to a builtin module.
- **isDependency** (0x80) - indicates that the program is a dependency module.
- **macroException** (0x100) - indicates that a macro exception has occurred.

rtti::**bitfield context_category_flags**

Flags which specify type of the *Context*.

Fields

- **dead** (0x1) - indicates that the context is dead.

- **debug_context** (0x2) - indicates that the context is a debug context.
- **thread_clone** (0x4) - indicates that the context is a thread clone.
- **job_clone** (0x8) - indicates that the context is a job clone.
- **opengl** (0x10) - indicates that the context is an opengl context.
- **debugger_tick** (0x20) - indicates that the context is a debugger tick.
- **debugger_attached** (0x40) - indicates that the context has a debugger attached.
- **macro_context** (0x80) - indicates that the context is a macro context (i.e. compiled macro module)
- **folding_context** (0x100) - indicates that the context is a folding context (used during compilation or optimization for the purposes of folding constants)
- **audio** (0x200) - indicates that the context is an audio context.

`rtti::bitfield TypeInfoFlags`

Flags which specify properties of the *TypeInfo* object (any rtti type).

Fields

- **ref** (0x1) - indicates that the type is a reference value.
- **refType** (0x2) - indicates that the type is a reference type.
- **canCopy** (0x4) - indicates that the type can be copied.
- **isPod** (0x8) - indicates that the type is a plain old data type.
- **isRawPod** (0x10) - indicates that the type is a raw plain old data type (without pointers or strings).
- **isConst** (0x20) - indicates that the type is a const type.
- **isTemp** (0x40) - indicates that the type is a temporary type.
- **isImplicit** (0x80) - indicates that the type is an implicit type.
- **refValue** (0x100) - indicates that the type is a reference value.
- **hasInitValue** (0x200) - indicates that the type has an initial value.
- **isSmartPtr** (0x400) - indicates that the type is a smart pointer.
- **isSmartPtrNative** (0x800) - indicates that the type is a smart pointer native (smart_ptr_raw)
- **isHandled** (0x1000) - indicates that the type is a handled type (annotation)
- **heapGC** (0x2000) - indicates that the type needs marking by the garbage collector.
- **stringHeapGC** (0x4000) - indicates that the type needs marking of strings by the garbage collector.
- **lockCheck** (0x8000) - indicates that the type needs lock checking.
- **isPrivate** (0x10000) - indicates that the type is private.

`rtti::bitfield StructInfoFlags`

Flags which represent properties of the *StructInfo* object (rtti object which represents structure type).

Fields

- **_class** (0x1) - This structure is a class.

- **_lambda** (0x2) - This structure is a lambda.
- **heapGC** (0x4) - This structure needs marking by the garbage collector.
- **stringHeapGC** (0x8) - This structure needs marking of strings by the garbage collector.
- **lockCheck** (0x10) - This structure needs lock checking.

`rtti::bitfield ModuleFlags`

Flags which represent the module's state.

Fields

- **builtIn** (0x1) - This module is built-in.
- **promoted** (0x2) - This module is promoted to a builtin module.
- **isPublic** (0x4) - This module is public.
- **isModule** (0x8) - This module is a module.
- **isSolidContext** (0x10) - This module is a solid context (can't be called from other contexts via pinvoke, global variables are cemented at locations)
- **fromExtraDependency** (0x20) - This module is from an extra dependency.
- **doNotAllowUnsafe** (0x40) - This module does not allow unsafe code.
- **wasParsedNameless** (0x80) - This module was parsed nameless.
- **visibleEverywhere** (0x100) - This module is visible everywhere.
- **skipLockCheck** (0x200) - This module skips lock checking.
- **allowPodInscope** (0x400) - This module allows pod inscope.

`rtti::bitfield AnnotationDeclarationFlags`

Flags which represent properties of the *AnnotationDeclaration* object.

Fields

- **inherited** (0x1) - Indicates that the annotation is inherited.

`rtti::bitfield SimFunctionFlags`

properties of the *SimFunction* object.

Fields

- **aot** (0x1) - Function is compiled ahead-of-time.
- **fastcall** (0x2) - Function uses fastcall calling convention.
- **builtin** (0x4) - Function is a builtin.
- **jit** (0x8) - Function is JIT-compiled.
- **unsafe** (0x10) - Function is unsafe.
- **cmres** (0x20) - Function returns via caller-managed return space.
- **pinvoke** (0x40) - Function uses platform invoke for external calls.

`rtti::bitfield LocalVariableInfoFlags`

properties of the *LocalVariableInfo* object.

Fields

- **cmres** (0x1) - Variable is a caller-managed return space variable.

`rtti::variant RttiValue`

Variant type which represents value of any annotation arguments and variable annotations.

Variants

- **tBool** : bool - boolean value
- **tInt** : int - integer value
- **tUInt** : uint - unsigned integer value
- **tInt64** : int64 - 64-bit integer value
- **tUInt64** : uint64 - 64-bit unsigned integer value
- **tFloat** : float - floating point value
- **tDouble** : double - double precision floating point value
- **tString** : string - string value
- **nothing** : any - no value

`rtti::FileAccessPtr = smart_ptr<FileAccess>`

Type alias for `smart_ptr<FileAccess>` — a reference-counted pointer to a `FileAccess` object, used as the standard way to pass file access to the compiler.

10.1.2 Constants

`rtti::FUNCINFO_INIT = 0x1`

Bit flag constant on `FuncInfo.flags` indicating that the function runs during `Context` initialization (`[init]` attribute).

`rtti::FUNCINFO_BUILTIN = 0x2`

Bit flag constant on `FuncInfo.flags` indicating that the function is a built-in (C++-bound) function rather than a daslang-defined one.

`rtti::FUNCINFO_PRIVATE = 0x4`

Bit flag constant on `FuncInfo.flags` indicating that the function has `[private]` visibility and cannot be called from other modules.

`rtti::FUNCINFO_SHUTDOWN = 0x8`

Bit flag constant on `FuncInfo.flags` indicating that the function runs during `Context` shutdown (`[finalize]` attribute).

`rtti::FUNCINFO_LATE_INIT = 0x20`

Bit flag constant on `FuncInfo.flags` indicating the function uses late initialization with a custom init order (`[init(order)]` attribute).

10.1.3 Enumerations

`rtti::CompilationError`

Enumeration which represents error type for each of the errors which compiler returns and various stages.

Values

- **unspecified** = 0 - Unspecified error.
- **mismatching_parentheses** = 10001 - Mismatching parentheses.
- **mismatching_curly_bracers** = 10002 - Mismatching curly braces.
- **string_constant_exceeds_file** = 10003 - String constant exceeds file.
- **string_constant_exceeds_line** = 10004 - String constant exceeds line.
- **unexpected_close_comment** = 10005 - Unexpected close comment.
- **integer_constant_out_of_range** = 10006 - Integer constant out of range.
- **comment_contains_eof** = 10007 - Comment contains EOF (end of file).
- **invalid_escape_sequence** = 10008 - Invalid escape sequence.
- **invalid_line_directive** = 10009 - Invalid line directive.
- **syntax_error** = 20000 - Syntax error, usually invalid grammar.
- **malformed_ast** = 20001 - Malformed AST.
- **invalid_type** = 30101 - Invalid type.
- **invalid_return_type** = 30102 - Invalid return type.
- **invalid_argument_type** = 30103 - Invalid argument type.
- **invalid_structure_field_type** = 30104 - Invalid structure field type.
- **invalid_array_type** = 30105 - Invalid array type.
- **invalid_table_type** = 30106 - Invalid table type.
- **invalid_argument_count** = 30107 - Invalid argument count.
- **invalid_variable_type** = 30108 - Invalid variable type.
- **invalid_new_type** = 30109 - Invalid new type.
- **invalid_index_type** = 30110 - Invalid index type.
- **invalid_annotation** = 30111 - Invalid annotation.
- **invalid_swizzle_mask** = 30112 - Invalid swizzle mask.
- **invalid_initialization_type** = 30113 - Invalid initialization type.
- **invalid_with_type** = 30114 - Invalid with type.
- **invalid_override** = 30115 - Invalid override.
- **invalid_name** = 30116 - Invalid name.
- **invalid_array_dimension** = 30117 - Invalid array dimension.
- **invalid_iteration_source** = 30118 - Invalid iteration source.
- **invalid_loop** = 30119 - Invalid loop.

- **invalid_label** = 30120 - Invalid label.
- **invalid_enumeration** = 30121 - Invalid enumeration.
- **invalid_option** = 30122 - Invalid or unsupported option.
- **invalid_member_function** = 30123 - Invalid member function.
- **function_already_declared** = 30201 - Function already declared.
- **argument_already_declared** = 30202 - Argument already declared.
- **local_variable_already_declared** = 30203 - Local variable already declared.
- **global_variable_already_declared** = 30204 - Global variable already declared.
- **structure_field_already_declared** = 30205 - Structure field already declared.
- **structure_already_declared** = 30206 - Structure already declared.
- **structure_already_has_initializer** = 30207 - Structure already has initializer.
- **enumeration_already_declared** = 30208 - Enumeration already declared.
- **enumeration_value_already_declared** = 30209 - Enumeration value already declared.
- **type_alias_already_declared** = 30210 - Type alias already declared.
- **field_already_initialized** = 30211 - Field already initialized.
- **type_not_found** = 30301 - Type not found.
- **structure_not_found** = 30302 - Structure not found.
- **operator_not_found** = 30303 - Operator not found.
- **function_not_found** = 30304 - Function not found.
- **variable_not_found** = 30305 - Variable not found.
- **handle_not_found** = 30306 - Handle not found.
- **annotation_not_found** = 30307 - Annotation not found.
- **enumeration_not_found** = 30308 - Enumeration not found.
- **enumeration_value_not_found** = 30309 - Enumeration value not found.
- **type_alias_not_found** = 30310 - Type alias not found.
- **bitfield_not_found** = 30311 - Bitfield not found.
- **cant_initialize** = 30401 - Can't initialize.
- **cant_dereference** = 30501 - Can't dereference (not a pointer or dereferencable type).
- **cant_index** = 30502 - Can't index (not an array, table, or indexable type).
- **cant_get_field** = 30503 - Can't get field (not a structure or table type).
- **cant_write_to_const** = 30504 - Can't write to const.
- **cant_move_to_const** = 30505 - Can't move to const.
- **cant_write_to_non_reference** = 30506 - Can't write to non-reference.
- **cant_copy** = 30507 - Can't copy.
- **cant_move** = 30508 - Can't move.
- **cant_pass_temporary** = 30509 - Can't pass temporary value to non-temporary parameter.

- **condition_must_be_bool** = 30601 - Condition must be boolean.
- **condition_must_be_static** = 30602 - Condition must be static (for 'static_if' and 'static_elif')
- **cant_pipe** = 30701 - Can't pipe (invalid left-hand side or right-hand side).
- **invalid_block** = 30801 - Invalid block.
- **return_or_break_in_finally** = 30802 - Return or break in finally section is not allowed.
- **module_not_found** = 30901 - Module not found.
- **module_already_has_a_name** = 30902 - Module already has a name.
- **cant_new_handle** = 31001 - Can't new handled type.
- **bad_delete** = 31002 - Bad delete.
- **cant_infer_generic** = 31100 - Can't infer generic.
- **cant_infer_missing_initializer** = 31101 - Can't infer missing initializer.
- **cant_infer_mismatching_restrictions** = 31102 - Can't infer mismatching restrictions.
- **invalid_cast** = 31200 - Invalid cast.
- **incompatible_cast** = 31201 - Incompatible cast.
- **unsafe** = 31300 - Unsafe operation.
- **index_out_of_range** = 31400 - Index out of range.
- **expecting_return_value** = 32101 - Expecting return value.
- **not_expecting_return_value** = 32102 - Not expecting return value (void function or block).
- **invalid_return_semantics** = 32103 - Invalid return semantics.
- **invalid_yield** = 32104 - Invalid yield.
- **typeinfo_reference** = 39901 - 'typeinfo' error, the type is a reference.
- **typeinfo_auto** = 39902 - 'typeinfo' error, the type is auto.
- **typeinfo_undefined** = 39903 - 'typeinfo' error, the type is undefined.
- **typeinfo_dim** = 39904 - 'typeinfo' error, the type is not a static array.
- **typeinfo_macro_error** = 39905 - Macro returned error.
- **static_assert_failed** = 40100 - Static assert failed.
- **run_failed** = 40101 - Run failed (attempt of folding constant function without side-effects failed)
- **annotation_failed** = 40102 - Annotation throw panic during compile time.
- **concept_failed** = 40103 - Concept throw panic during compile time.
- **not_all_paths_return_value** = 40200 - Not all paths return value.
- **assert_with_side_effects** = 40201 - Assert with side effects.
- **only_fast_aot_no_cpp_name** = 40202 - Only fast AOT no C++ name.
- **aot_side_effects** = 40203 - AOT side effects.
- **no_global_heap** = 40204 - No global heap is specified, but program requests it.
- **no_global_variables** = 40205 - No global variables are allowed in this context.
- **unused_function_argument** = 40206 - Unused function argument.

- **unsafe_function** = 40207 - Unsafe function.
- **too_many_infer_passes** = 41000 - Too many infer passes.
- **missing_node** = 50100 - Missing simulation node.

`rtti::ConstMatters`

Yes or no flag which indicates if constant flag of the type matters (during comparison).

Values

- **no** = 0 - const does not matter, when comparing types.
- **yes** = 1 - const matters, when comparing types.

`rtti::RefMatters`

Yes or no flag which indicates if reference flag of the type matters (during comparison).

Values

- **no** = 0 - Ref does not matter, when comparing types.
- **yes** = 1 - Ref matters, when comparing types.

`rtti::TemporaryMatters`

Yes or no flag which indicates if temporary flag of the type matters (during comparison).

Values

- **no** = 0 - Temporary does not matter, when comparing types.
- **yes** = 1 - Temporary matters, when comparing types.

`rtti::Type`

One of the fundamental (base) types of any type object.

Values

- **none** = 0 - No type specified (not void, not anything).
- **autoinfer** = 1 - Auto-inferred type (auto)
- **alias** = 2 - Type alias.
- **option** = 3 - Optional type (foo|bar|…)
- **typeDecl** = 4 - Type declaration typedecl(expr…)
- **typeMacro** = 5 - Type macro. $type_macro_name(args…)
- **fakeContext** = 6 - Fake context type (used for internal purposes to pass context as argument to C++ functions)
- **fakeLineInfo** = 7 - Fake line info type (used for internal purposes to pass line info as argument to C++ functions)
- **anyArgument** = 8 - Any argument type (only available for C++ interop functions, TypeInfo is provided)
- **tVoid** = 9 - Void type.
- **tBool** = 10 - Boolean type.
- **tInt8** = 11 - 8-bit integer type.

- **tUInt8** = 12 - 8-bit unsigned integer type.
- **tInt16** = 13 - 16-bit integer type.
- **tUInt16** = 14 - 16-bit unsigned integer type.
- **tInt64** = 15 - 64-bit integer type.
- **tUInt64** = 16 - 64-bit unsigned integer type.
- **tInt** = 17 - 32-bit integer type.
- **tInt2** = 18 - vector of 2 32-bit integers.
- **tInt3** = 19 - vector of 3 32-bit integers.
- **tInt4** = 20 - vector of 4 32-bit integers.
- **tUInt** = 21 - 32-bit unsigned integer type.
- **tUInt2** = 22 - vector of 2 32-bit unsigned integers.
- **tUInt3** = 23 - vector of 3 32-bit unsigned integers.
- **tUInt4** = 24 - vector of 4 32-bit unsigned integers.
- **tFloat** = 25 - 32-bit floating point type.
- **tFloat2** = 26 - vector of 2 32-bit floating point numbers.
- **tFloat3** = 27 - vector of 3 32-bit floating point numbers.
- **tFloat4** = 28 - vector of 4 32-bit floating point numbers.
- **tDouble** = 29 - 64-bit floating point type.
- **tRange** = 30 - 32-bit range type, similar to vector of two 32-bit integers.
- **tURange** = 31 - 32-bit unsigned range type, similar to vector of two 32-bit unsigned integers.
- **tRange64** = 32 - 64-bit range type, similar to vector of two 64-bit integers.
- **tURange64** = 33 - 64-bit unsigned range type, similar to vector of two 64-bit unsigned integers.
- **tString** = 34 - String type.
- **tStructure** = 35 - Structure type.
- **tHandle** = 36 - Handle type (C++ type specified via annotation).
- **tEnumeration** = 37 - 32-bit enumeration type.
- **tEnumeration8** = 38 - 8-bit enumeration type.
- **tEnumeration16** = 39 - 16-bit enumeration type.
- **tEnumeration64** = 40 - 64-bit enumeration type.
- **tBitfield** = 41 - 32-bit bitfield type.
- **tBitfield8** = 42 - 8-bit bitfield type.
- **tBitfield16** = 43 - 16-bit bitfield type.
- **tBitfield64** = 44 - 64-bit bitfield type.
- **tPointer** = 45 - Pointer type.
- **tFunction** = 46 - Function type.
- **tLambda** = 47 - Lambda type.

- **tIterator** = 48 - Iterator type.
- **tArray** = 49 - Array type.
- **tTable** = 50 - Table type.
- **tBlock** = 51 - Block type.
- **tTuple** = 52 - Tuple type.
- **tVariant** = 53 - Variant type.

10.1.4 Handled structures

`rtti::CodeOfPolicies`

Fields

- **aot** : bool - Object which holds compilation and simulation settings and restrictions.
- **aot_lib** : bool - Whether ahead-of-time compilation is enabled.
- **standalone_context** : bool - AOT library mode.
- **aot_module** : bool - Whether standalone context AOT compilation is enabled.
- **aot_macros** : bool - Specifies to AOT if we are compiling a module, or a final program.
- **paranoid_validation** : bool - Enables AOT of macro code (like 'qmacro_block' etc).
- **cross_platform** : bool - Whether paranoid validation is enabled (extra checks, no optimizations).
- **aot_result** : *das_string* - Whether cross-platform AOT is enabled (if not, we generate code for the current platform).
- **completion** : bool - File name for AOT output (if not set, we generate a temporary file).
- **export_all** : bool - If we are in code completion mode.
- **serialize_main_module** : bool - Export all functions and global variables.
- **keep_alive** : bool - If not set, we recompile main module each time.
- **very_safe_context** : bool - Keep context alive after main function.
- **always_report_candidates_threshold** : int - Whether to use very safe context (delete of data is delayed, to avoid table[foo]=table[bar] lifetime bugs).
- **max_infer_passes** : int - Threshold for reporting candidates for function calls. If less than this number, we always report them.
- **stack** : uint - Maximum number of inference passes.
- **intern_strings** : bool - Stack size.
- **persistent_heap** : bool - Whether to intern strings.
- **multiple_contexts** : bool - Whether to use persistent heap (or linear heap).
- **heap_size_hint** : uint - Whether multiple contexts are allowed (pinvokes between contexts).
- **string_heap_size_hint** : uint - Heap size hint.
- **solid_context** : bool - String heap size hint.
- **macro_context_persistent_heap** : bool - Whether to use solid context (global variables are cemented at locations, can't be called from other contexts via pinvoke).

- **macro_context_collect** : bool - Whether macro context uses persistent heap.
- **max_static_variables_size** : uint64 - Whether macro context does garbage collection.
- **max_heap_allocated** : uint64 - Maximum size of static variables.
- **max_string_heap_allocated** : uint64 - Maximum heap allocated.
- **rtti** : bool - Maximum string heap allocated.
- **unsafe_table_lookup** : bool - Whether to enable RTTI.
- **relaxed_pointer_const** : bool - Whether to allow unsafe table lookups (via [] operator).
- **version_2_syntax** : bool - Whether to relax pointer constness rules.
- **gen2_make_syntax** : bool - Allows use of version 2 syntax.
- **relaxed_assign** : bool - Whether to use gen2 make syntax.
- **no_unsafe** : bool - Allows relaxing of the assignment rules.
- **local_ref_is_unsafe** : bool - Disables all unsafe operations.
- **no_global_variables** : bool - Local references are considered unsafe.
- **no_global_variables_at_all** : bool - Disallows global variables in this context (except for generated).
- **no_global_heap** : bool - Disallows global variables at all in this context.
- **only_fast_aot** : bool - Disallows global heap in this context.
- **aot_order_side_effects** : bool - Only fast AOT, no C++ name generation.
- **no_unused_function_arguments** : bool - Whether to consider side effects during AOT ordering.
- **no_unused_block_arguments** : bool - Errors on unused function arguments.
- **allow_block_variable_shadowing** : bool - Errors on unused block arguments.
- **allow_local_variable_shadowing** : bool - Allows block variable shadowing.
- **allow_shared_lambda** : bool - Allows local variable shadowing.
- **ignore_shared_modules** : bool - Allows shared lambdas.
- **default_module_public** : bool - Ignore shared modules during compilation.
- **no_deprecated** : bool - Default module mode is public.
- **no_aliasing** : bool - Disallows use of deprecated features.
- **strict_smart_pointers** : bool - Disallows aliasing (if aliasing is allowed, temporary lifetimes are extended).
- **no_init** : bool - Enables strict smart pointer checks.
- **strict_unsafe_delete** : bool - Disallows use of ‘init’ in structures.
- **no_members_functions_in_struct** : bool - Enables strict unsafe delete checks.
- **no_local_class_members** : bool - Disallows member functions in structures.
- **report_invisible_functions** : bool - Disallows local class members.
- **report_private_functions** : bool - Report invisible functions.
- **strict_properties** : bool - Report private functions.

- **no_optimizations** : bool - Enables strict property checks.
- **fail_on_no_aot** : bool - Disables all optimizations.
- **fail_on_lack_of_aot_export** : bool - Fails compilation if AOT is not available.
- **log_compile_time** : bool - Fails compilation if AOT export is not available.
- **log_total_compile_time** : bool - Log compile time.
- **no_fast_call** : bool - Log total compile time.
- **scoped_stack_allocator** : bool - Disables fast call optimization.
- **force_inscope_pod** : bool - Reuse stack memory after variables go out of scope.
- **log_inscope_pod** : bool - Force in-scope for POD-like types.
- **debugger** : bool - Log in-scope for POD-like types.
- **debug_infer_flag** : bool - Enables debugger support.
- **debug_module** : *das_string* - Enables debug inference flag.
- **profiler** : bool - Sets debug module (module which will be loaded when IDE connects).
- **profile_module** : *das_string* - Enables profiler support.
- **threadlock_context** : bool - Sets profile module (module which will be loaded when profiler connects).
- **jit_enabled** : bool - Enables threadlock context.
- **jit_module** : *das_string* - JIT enabled - if enabled, JIT will be used to compile code at runtime.
- **jit_jit_all_functions** : bool - JIT module - module loaded when -jit is specified.
- **jit_debug_info** : bool - JIT all functions - if enabled, JIT will compile all functions in the module.
- **jit_use_dll_mode** : bool - JIT debug info - if enabled, JIT will generate debug info for JIT compiled code.
- **emit_prologue** : bool - JIT dll mode - if enabled, JIT will generate DLL's into JIT output folder and load them from there.
- **jit_output_folder** : *das_string* - JIT output folder (where JIT compiled code will be stored).
- **jit_opt_level** : int - JIT optimization level for compiled code (0-3).
- **jit_size_level** : int - JIT size optimization level for compiled code (0-3).
- **jit_path_to_shared_lib** : *das_string* - Path to shared library, which is used in JIT.
- **jit_path_to_linker** : *das_string* - Path to linker, which is used in JIT.

`rtti::`**`FileInfo`**

Information about a single file stored in the *FileAccess* object.

Fields

- **name** : *das_string* - File name.
- **tabSize** : int - Tab size for this file.

`rtti::LineInfo`

Information about a section of the file stored in the *FileAccess* object.

Fields

- **fileInfo** : *FileInfo*? - File information object.
- **column** : uint - Column number (1-based).
- **line** : uint - Line number (1-based).
- **last_column** : uint - Last column number (1-based).
- **last_line** : uint - Last line number (1-based).

`rtti::Context`

`Context.getInitSemanticHash() : uint64()`

Property-like accessor that returns the `uint64` semantic hash of the initialization code for the given `Context`, useful for detecting code changes.

`Context.totalFunctions() : int()`

Property-like accessor that returns the total number of registered `SimFunction` entries in the given `Context`.

`Context.totalVariables() : int()`

Property-like accessor that returns the total number of global variables registered in the given `Context`.

`Context.getCodeAllocatorId() : uint64()`

Property-like accessor that returns a non-persistent unique integer ID of the code (node) allocator associated with the given `Context`.

Properties

- **getInitSemanticHash** : uint64
- **totalFunctions** : int
- **totalVariables** : int
- **getCodeAllocatorId** : uint64

Object which holds single Daslang Context. Context is the result of the simulation of the Daslang program.

Fields

- **breakOnException** : bool - Calls breakpoint when exception is thrown.
- **alwaysErrorOnException** : bool - Always error on exception.
- **alwaysStackWalkOnException** : bool - Always stack walk on exception.
- **name** : *das_string* - Context name.
- **category** : *context_category_flags* - Context category flags.
- **exceptionAt** : *LineInfo* - Exception at line info.
- **exception** : string - Exception message.
- **last_exception** : string - Last exception message.
- **contextMutex** : *recursive_mutex*? - Context mutex.

rtti::**Error**

Object which holds information about compilation error or exception.

Fields

- **what** : *das_string* - What error is about.
- **extra** : *das_string* - Extra information about error.
- **fixme** : *das_string* - Fixme information about error.
- **at** : *LineInfo* - Line info where error occurred.
- **cerr** : *CompilationError* - Compilation error.

rtti::**FileAccess**

Object which holds collection of files as well as means to access them (Project).

rtti::**Module**

Collection of types, aliases, functions, classes, macros etc under a single namespace.

Fields

- **name** : *das_string* - Module name.
- **fileName** : *das_string* - Module file name.
- **moduleFlags** : *ModuleFlags* - Module flags.

rtti::**ModuleGroup**

Collection of modules.

rtti::**AnnotationArgument**

Single argument of the annotation, typically part of the *AnnotationArgumentList*.

Fields

- **basicType** : *Type* - type of the argument value
- **name** : *das_string* - argument name
- **sValue** : *das_string* - string value
- **iValue** : int - integer value
- **fValue** : float - float value
- **bValue** : bool - boolean value
- **at** : *LineInfo* - line info where the argument is defined

rtti::**Program**

Program.**getThisModule() : Module?**()

Property-like accessor that returns the `Module` pointer for the module currently being inferred in the given `Program`.

Program.**getDebugger() : bool**()

Property-like accessor that returns `true` if the debugger is attached and enabled for the given `Program`.

Properties

- **getThisModule** : *Module*?
- **getDebugger** : bool

Object representing full information about Daslang program during and after compilation (but not the simulated result of the program).

Fields

- **thisNamespace** : *das_string* - namespace of the program
- **thisModuleName** : *das_string* - module name
- **totalFunctions** : int - total number of functions in the program
- **totalVariables** : int - total number of variables in the program
- **errors** : vector<Error> - compilation errors and exceptions
- **globalStringHeapSize** : uint - size of the global string heap
- **initSemanticHashWithDep** : uint64 - initial semantic hash with dependencies
- **flags** : *ProgramFlags* - program flags
- **_options** : *AnnotationArgumentList* - program options
- **policies** : *CodeOfPolicies* - code of policies

`rtti::`**`Annotation`**

`Annotation.`**`isTypeAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is a `TypeAnnotation` (defines a handled type).

`Annotation.`**`isBasicStructureAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is a `BasicStructureAnnotation`, which exposes C++ struct fields to daslang.

`Annotation.`**`isStructureAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is a structure annotation (applied to struct declarations).

`Annotation.`**`isStructureTypeAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is a `StructureTypeAnnotation`, which binds a C++ class as a daslang handled struct.

`Annotation.`**`isFunctionAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is a `FunctionAnnotation` (applied to functions).

`Annotation.`**`isEnumerationAnnotation() : bool`**`()`

Property-like accessor that returns `true` if the given `Annotation` is an `EnumerationAnnotation`.

Properties

- **isTypeAnnotation** : bool
- **isBasicStructureAnnotation** : bool
- **isStructureAnnotation** : bool
- **isStructureTypeAnnotation** : bool
- **isFunctionAnnotation** : bool
- **isEnumerationAnnotation** : bool

Handled type or macro.

Fields

- **name** : *das_string* - name of the annotation
- **cppName** : *das_string* - name of the associated C++ type or macro
- **_module** : *Module*? - module where the annotation is defined

`rtti::`**`AnnotationDeclaration`**

Annotation declaration, its location, and arguments.

Fields

- **annotation** : smart_ptr< *Annotation*> - pointer to the handled annotation object.
- **arguments** : *AnnotationArgumentList* - list of annotation arguments.
- **at** : *LineInfo* - line information where the annotation is defined.
- **flags** : *AnnotationDeclarationFlags* - flags associated with the annotation.

`rtti::`**`TypeAnnotation`**

`TypeAnnotation.`**`is_any_vector() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` wraps any C++ vector-like container (e.g., `std::vector`).

`TypeAnnotation.`**`canMove() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` supports move semantics.

`TypeAnnotation.`**`canCopy() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` supports copy semantics.

`TypeAnnotation.`**`canClone() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` supports the clone operation.

`TypeAnnotation.`**`isPod() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` is a POD (plain old data) type — no constructor, destructor, or special semantics.

`TypeAnnotation.`**`isRawPod() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` is a raw POD type — a basic value type excluding pointers and strings.

`TypeAnnotation.`**`isRefType() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` is always passed by reference, or is itself a reference type.

`TypeAnnotation.`**`hasNonTrivialCtor() : bool`**`()`

Property-like accessor that returns `true` if the given `TypeAnnotation` has a non-trivial constructor (requires explicit initialization).

TypeAnnotation.**hasNonTrivialDtor() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` has a non-trivial destructor (requires explicit finalization).

TypeAnnotation.**hasNonTrivialCopy() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` has non-trivial copy semantics (i.e., a custom copy constructor).

TypeAnnotation.**canBePlacedInContainer() : bool**()

Property-like accessor that returns `true` if values of the given `TypeAnnotation` can be stored inside arrays, tables, or other containers.

TypeAnnotation.**isLocal() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` can be used as a local variable type within a function.

TypeAnnotation.**canNew() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` supports heap allocation via `new`.

TypeAnnotation.**canDelete() : bool**()

Property-like accessor that returns `true` if values of the given `TypeAnnotation` can be explicitly deleted.

TypeAnnotation.**needDelete() : bool**()

Property-like accessor that returns `true` if values of the given `TypeAnnotation` require explicit `delete` to free resources.

TypeAnnotation.**canDeletePtr() : bool**()

Property-like accessor that returns `true` if a pointer to the given `TypeAnnotation` type can be explicitly deleted.

TypeAnnotation.**isIterable() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` supports iteration via `for`.

TypeAnnotation.**isShareable() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` can be shared across multiple `Context` objects.

TypeAnnotation.**isSmart() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` represents a `smart_ptr` managed type.

TypeAnnotation.**avoidNullPtr() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` requires pointers to its type to be non-null (i.e., must be initialized on creation).

TypeAnnotation.**sizeOf() : uint64**()

Property-like accessor that returns the size in bytes of the type described by the given `TypeAnnotation`.

TypeAnnotation.**alignOf() : uint64**()

Property-like accessor that returns the memory alignment requirement (in bytes) of the type described by the given `TypeAnnotation`.

Properties

- **is_any_vector** : bool
- **canMove** : bool
- **canCopy** : bool
- **canClone** : bool
- **isPod** : bool
- **isRawPod** : bool
- **isRefType** : bool
- **hasNonTrivialCtor** : bool
- **hasNonTrivialDtor** : bool
- **hasNonTrivialCopy** : bool
- **canBePlacedInContainer** : bool
- **isLocal** : bool
- **canNew** : bool
- **canDelete** : bool
- **needDelete** : bool
- **canDeletePtr** : bool
- **isIterable** : bool
- **isShareable** : bool
- **isSmart** : bool
- **avoidNullPtr** : bool
- **sizeOf** : uint64
- **alignOf** : uint64

Handled type.

Fields

- **name** : *das_string* - name of the type annotation
- **cppName** : *das_string* - name of the associated C++ type
- **_module** : *Module*? - module where the annotation is defined

`rtti::`**`BasicStructureAnnotation`**

`BasicStructureAnnotation.`**`fieldCount() : int`**`()`

Property-like accessor that returns the number of fields declared in the given `BasicStructureAnnotation`.

Properties

- **fieldCount** : int

Handled type which represents a structure-like annotation for exposing C++ types to daslang.

Fields

- **name** : *das_string* - Name of the annotation
- **cppName** : *das_string* - C++ class name used in AOT code generation

rtti::**EnumValueInfo**

Single element of enumeration, its name and value.

Fields

- **name** : string - name of the enumeration value
- **value** : int64 - value of the enumeration value

rtti::**EnumInfo**

Type object which represents enumeration.

Fields

- **name** : string - name of the enumeration
- **module_name** : string - module where the enumeration is defined
- **fields** : *EnumValueInfo*?? - fields in the enumeration
- **count** : uint - number of fields in the enumeration
- **hash** : uint64 - hash of the enumeration

rtti::**StructInfo**

Type object which represents structure or class.

Fields

- **name** : string - name of the structure
- **module_name** : string - module where the structure is defined
- **fields** : *VarInfo*?? - fields in the structure
- **hash** : uint64 - hash of the structure
- **init_mnh** : uint64 - hash of the structure initializer
- **flags** : *StructInfoFlags* - flags associated with the structure
- **count** : uint - number of fields in the structure
- **size** : uint - size of the structure in bytes
- **firstGcField** : uint - index of the first GC field in the structure, i.e. field which requires garbage collection marking

rtti::**TypeInfo**

TypeInfo.**enumType() : EnumInfo?**()

Property-like accessor that returns the `EnumInfo` pointer describing the underlying enumeration for the given enum `TypeAnnotation`.

TypeInfo.**isRef() : bool**()

Property-like accessor that returns `true` if the given `TypeInfo` describes a reference (&) type.

TypeInfo.**isRefType() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` is always passed by reference, or is itself a reference type.

TypeInfo.**isRefValue() : bool**()

Property-like accessor that returns `true` if the given `TypeInfo` describes a ref-value type (boxed value accessed by reference).

TypeInfo.**canCopy() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` supports copy semantics.

TypeInfo.**isPod() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` is a POD (plain old data) type — no constructor, destructor, or special semantics.

TypeInfo.**isRawPod() : bool**()

Property-like accessor that returns `true` if the given `TypeAnnotation` is a raw POD type — a basic value type excluding pointers and strings.

TypeInfo.**isConst() : bool**()

Property-like accessor that returns `true` if the given `TypeInfo` describes a `const`-qualified type.

TypeInfo.**isTemp() : bool**()

Property-like accessor that returns `true` if the given `TypeInfo` describes a temporary (#) type that cannot be captured or stored.

TypeInfo.**isImplicit() : bool**()

Property-like accessor that returns `true` if the given `TypeInfo` describes an implicit (compiler-inferred) type.

TypeInfo.**annotation() : TypeAnnotation?**()

Property-like accessor that returns the `Annotation` pointer associated with the given `TypeInfo`.

TypeInfo.**annotation_or_name() : TypeAnnotation?**()

Property-like accessor that returns the annotation name if one exists, otherwise returns the raw type name from the given `TypeInfo`.

TypeInfo.**structType() : StructInfo?**()

Property-like accessor that returns the `StructInfo` pointer for the struct described by the given `TypeInfo`, or null if not a struct type.

Properties

- **enumType** : *EnumInfo*?
- **isRef** : bool
- **isRefType** : bool
- **isRefValue** : bool
- **canCopy** : bool
- **isPod** : bool
- **isRawPod** : bool
- **isConst** : bool
- **isTemp** : bool
- **isImplicit** : bool

- **annotation** : *TypeAnnotation*?
- **annotation_or_name** : *TypeAnnotation*?
- **structType** : *StructInfo*?

Object which represents any Daslang type.

Fields

- **firstType** : *TypeInfo*? - first type parameter
- **secondType** : *TypeInfo*? - second type parameter
- **argTypes** : *TypeInfo*?? - argument types list
- **argNames** : string? - argument names list
- **dim** : uint? - dimensions list
- **hash** : uint64 - hash of the type
- **_type** : *Type* - kind of the type
- **basicType** : *Type* - basic type category
- **flags** : *TypeInfoFlags* - flags associated with the type
- **size** : uint - size of the type in bytes
- **argCount** : uint - number of arguments in the type
- **dimSize** : uint - number of dimensions in the type

`rtti::`**`VarInfo`**

Object which represents variable declaration.

Fields

- **firstType** : *TypeInfo*? - first type parameter
- **secondType** : *TypeInfo*? - second type parameter
- **argTypes** : *TypeInfo*?? - argument types list
- **argNames** : string? - argument names list
- **hash** : uint64 - hash of the type
- **basicType** : *Type* - basic type category
- **flags** : *TypeInfoFlags* - flags associated with the type
- **size** : uint - size of the type in bytes
- **argCount** : uint - number of arguments in the type
- **dimSize** : uint - number of dimensions in the type
- **value** : any - value of the variable
- **sValue** : string - string value of the variable
- **name** : string - name of the variable
- **annotation_arguments** : *AnnotationArguments*? - annotation arguments
- **offset** : uint - offset of the variable in the structure

- **nextGcField** : uint - next garbage collection field in the structure, which requires garbage collection marking

`rtti::LocalVariableInfo`

Object which represents local variable declaration.

Fields

- **firstType** : *TypeInfo*? - first type parameter
- **secondType** : *TypeInfo*? - second type parameter
- **argTypes** : *TypeInfo*?? - argument types list
- **argNames** : string? - argument names list
- **hash** : uint64 - hash of the type
- **basicType** : *Type* - basic type category
- **flags** : *TypeInfoFlags* - flags associated with the type
- **size** : uint - size of the type in bytes
- **argCount** : uint - number of arguments in the type
- **dimSize** : uint - number of dimensions in the type
- **visibility** : *LineInfo* - visibility information
- **name** : string - name of the variable
- **stackTop** : uint - stack top offset
- **localFlags** : *LocalVariableInfoFlags* - local variable flags

`rtti::FuncInfo`

Object which represents function declaration.

Fields

- **name** : string - function name
- **cppName** : string - C++ name (for the builtin functions)
- **fields** : *VarInfo*?? - function parameters
- **result** : *TypeInfo*? - function result type
- **locals** : *LocalVariableInfo*?? - local variables (with visibility info)
- **globals** : *VarInfo*?? - accessed global variables
- **hash** : uint64 - hash of the function
- **flags** : uint - flags associated with the function
- **count** : uint - number of arguments in the function
- **stackSize** : uint - stack size in bytes
- **localCount** : uint - number of local variables
- **globalCount** : uint - number of accessed global variables

`rtti::SimFunction`

SimFunction.**lineInfo() : LineInfo const?**()

Property-like accessor that returns the `LineInfo` (source location) associated with the given function's `FuncInfo`.

Properties

- **lineInfo** : *LineInfo*?

Object which represents simulated function in the *Context*.

Fields

- **name** : string - original function name
- **mangledName** : string - mangled function name (with arguments and types encoded)
- **debugInfo** : *FuncInfo*? - pointer to the function debug info, if available
- **mangledNameHash** : uint64 - hash of the mangled function name
- **stackSize** : uint - stack size in bytes
- **flags** : *SimFunctionFlags* - flags associated with the function

rtti::**DebugInfoHelper**

Helper object which holds debug information about the simulated program.

Fields

- **rtti** : bool - The RTTI context pointer.

10.1.5 Typeinfo macros

rtti::**rtti_typeinfo**

Typeinfo macro that provides compile-time access to RTTI type information structures. Typeinfo macro rtti_typeinfo

10.1.6 Handled types

rtti::**recursive_mutex**

Handled type wrapping a system `std::recursive_mutex`, used with `lock_mutex` for thread-safe access to shared data across contexts.

rtti::**AnnotationArguments**

Handled type representing a collection of annotation arguments, typically the raw argument list parsed from an annotation declaration.

rtti::**AnnotationArgumentList**

Handled type representing an ordered list of annotation arguments and properties, providing indexed and named access to argument entries.

rtti::**AnnotationList**

Handled type representing all annotations attached to a single object (function, structure, or variable), iterable via `each`.

10.1.7 Initialization and finalization

- *CodeOfPolicies () : CodeOfPolicies*
- *LineInfo () : LineInfo*
- *LineInfo (arg0: FileInfo?; arg1: int; arg2: int; arg3: int; arg4: int) : LineInfo*
- *RttiValue_nothing () : auto*
- *using (arg0: block<(CodeOfPolicies):void>)*
- *using (arg0: block<(ModuleGroup):void>)*
- *using (arg0: block<(recursive_mutex):void>)*

`rtti::`**`CodeOfPolicies() : CodeOfPolicies`**`()`

Constructs a default-initialized `CodeOfPolicies` structure, which controls compiler behavior and optimization settings.

LineInfo

`rtti::`**`LineInfo() : LineInfo`**`()`

Constructs a default-initialized `LineInfo` structure representing source file location (file, line, column).

`rtti::`**`LineInfo(arg0: FileInfo?; arg1: int; arg2: int; arg3: int; arg4: int) : LineInfo`**`()`

`rtti::`**`RttiValue_nothing() : auto`**`()`

Constructs an `RttiValue` variant set to the `nothing` alternative, representing an absent or void value.

using

`rtti::`**`using`**`(`*arg0: block<(CodeOfPolicies):void>*`)`

Creates a temporary RTTI helper object (e.g., `Program`, `DebugInfoHelper`) scoped to the given block, automatically finalized on block exit.

Arguments

- **arg0** : block<(*CodeOfPolicies*):void> implicit

`rtti::`**`using`**`(`*arg0: block<(ModuleGroup):void>*`)`

`rtti::`**`using`**`(`*arg0: block<(recursive_mutex):void>*`)`

10.1.8 Type access

- *arg_names (info: TypeInfo) : auto*
- *arg_names (info: VarInfo) : auto*
- *arg_types (info: VarInfo) : auto*
- *arg_types (info: TypeInfo) : auto*
- *builtin_is_same_type (a: TypeInfo const?; b: TypeInfo const?; refMatters: RefMatters; cosntMatters: ConstMatters; tempMatters: TemporaryMatters; topLevel: bool) : bool*
- *each_dim (info: TypeInfo) : auto*
- *each_dim (info: VarInfo) : auto*
- *get_das_type_name (type: Type) : string*
- *get_dim (typeinfo: VarInfo; index: int) : int*
- *get_dim (typeinfo: TypeInfo; index: int) : int*
- *get_type_align (type: TypeInfo?) : int*
- *get_type_size (type: TypeInfo?) : int*
- *is_compatible_cast (from: StructInfo const?; to: StructInfo const?) : bool*
- *is_compatible_cast (a: StructInfo; b: StructInfo) : auto*
- *is_same_type (a: TypeInfo; b: TypeInfo; refMatters: RefMatters = RefMatters.yes; constMatters: ConstMatters = ConstMatters.yes; temporaryMatters: TemporaryMatters = TemporaryMatters.yes; topLevel: bool = true) : auto*

arg_names

`rtti::`**`arg_names(info: TypeInfo) : auto`**`()`

Iterates through the argument names of an RTTI type, yielding each name as a `string` — used for inspecting function or call-site parameter names.

Arguments

- **info** : *TypeInfo*

`rtti::`**`arg_names(info: VarInfo) : auto`**`()`

---

arg_types

`rtti::`**`arg_types(info: VarInfo) : auto`**`()`

Iterates through the argument types of an RTTI type, yielding each element as a `TypeInfo` pointer — used for inspecting function or call-site parameter types.

Arguments

- **info** : *VarInfo*

rtti::**arg_types(info: TypeInfo) : auto**()

---

rtti::**builtin_is_same_type(a: TypeInfo const?; b: TypeInfo const?; refMatters: RefMatters; cosntMatters**

Returns `true` if two `TypeInfo` pointers describe the same type, with flags controlling whether ref, const, temp, and other qualifiers are included in the comparison.

Arguments

- **a** : *TypeInfo*? implicit
- **b** : *TypeInfo*? implicit
- **refMatters** : *RefMatters*
- **cosntMatters** : *ConstMatters*
- **tempMatters** : *TemporaryMatters*
- **topLevel** : bool

each_dim

rtti::**each_dim(info: TypeInfo) : auto**()

Iterates through the dimension sizes of a fixed-size array `TypeInfo`, yielding each `int` dimension value (e.g., `int[3][4]` yields 3 then 4).

Arguments

- **info** : *TypeInfo*

rtti::**each_dim(info: VarInfo) : auto**()

---

rtti::**get_das_type_name(type: Type) : string**()

Returns the canonical `string` name of the given `Type` enumeration value (e.g., `tInt` → `"int"`).

Arguments

- **type** : *Type*

get_dim

rtti::**get_dim(typeinfo: VarInfo; index: int) : int**()

Returns the dimension size (`int`) at the specified index for a fixed-size array type described by `TypeInfo`.

Arguments

- **typeinfo** : *VarInfo* implicit
- **index** : int

rtti::**get_dim(typeinfo: TypeInfo; index: int) : int**()

---

rtti::**get_type_align(type: TypeInfo?) : int**()

Returns the memory alignment (`int`, in bytes) of the type described by the given `TypeInfo`.

Arguments

- **type** : *TypeInfo*? implicit

rtti::**get_type_size(type: TypeInfo?) : int**()

Returns the size (`int`, in bytes) of the type described by the given `TypeInfo`.

Arguments

- **type** : *TypeInfo*? implicit

is_compatible_cast

rtti::**is_compatible_cast(from: StructInfo const?; to: StructInfo const?) : bool**()

Returns `true` if an object of type `from` (`StructInfo`) can be safely cast to type `to` (`StructInfo`), following the class hierarchy.

Arguments

- **from** : *StructInfo*? implicit
- **to** : *StructInfo*? implicit

rtti::**is_compatible_cast(a: StructInfo; b: StructInfo) : auto**()

`is_same_type(a: TypeInfo; b: TypeInfo; refMatters: RefMatters = RefMatters.yes; constMatters: ConstMatt`

Returns `true` if two `TypeInfo` objects describe the same type, with flags controlling comparison of qualifiers (ref, const, temp, etc.).

Arguments

- **a** : *TypeInfo*
- **b** : *TypeInfo*
- **refMatters** : *RefMatters*
- **constMatters** : *ConstMatters*
- **temporaryMatters** : *TemporaryMatters*
- **topLevel** : bool

10.1.9 Rtti context access

- *class_info (cl: auto) : StructInfo const?*
- *context_for_each_function (blk: block<(info:FuncInfo):void>) : auto*
- *context_for_each_variable (blk: block<(info:VarInfo):void>) : auto*
- *get_function_by_mnh (context: Context; MNH: uint64) : function<():void>*
- *get_function_info (context: any; index: int) : FuncInfo*

- *get_function_info (context: Context; function: function<():void>) : FuncInfo const?*
- *get_line_info () : LineInfo*
- *get_line_info (depth: int) : LineInfo*
- *get_total_functions (context: Context) : int*
- *get_total_variables (context: Context) : int*
- *get_variable_info (context: any; index: int) : VarInfo*
- *get_variable_value (varInfo: VarInfo) : RttiValue*
- *this_context () : Context&*
- *type_info (cl: auto) : TypeInfo const?*
- *type_info (vinfo: VarInfo) : TypeInfo const?*
- *type_info (vinfo: LocalVariableInfo) : TypeInfo const?*

`rtti::`**`class_info(cl: auto) : StructInfo const?`**`()`

Returns a `StructInfo` pointer for the given class instance, enabling runtime introspection of its fields and annotations via RTTI.

Arguments

- **cl** : auto

`rtti::`**`context_for_each_function(blk: block<(info:FuncInfo):void>) : auto`**`()`

Iterates through all functions in the given `Context`, yielding a `FuncInfo` pointer for each registered function.

Arguments

- **blk** : block<(info: *FuncInfo*):void>

`rtti::`**`context_for_each_variable(blk: block<(info:VarInfo):void>) : auto`**`()`

Iterates through all global variables in the given `Context`, yielding a `VarInfo` pointer for each registered variable.

Arguments

- **blk** : block<(info: *VarInfo*):void>

`rtti::`**`get_function_by_mnh(context: Context; MNH: uint64) : function<():void>`**`()`

Returns a `SimFunction` pointer looked up by mangled name hash — an alternative form of `get_function_address`.

Arguments

- **context** : *Context* implicit
- **MNH** : uint64

get_function_info

`rtti::get_function_info(context: any; index: int) : FuncInfo()`

Returns the `FuncInfo` pointer for a function at the given index in the `Context`, providing access to its name, arguments, and return type.

Arguments

- **context** : any
- **index** : int

`rtti::get_function_info(context: Context; function: function<():void>) : FuncInfo const?()`

---

get_line_info

`rtti::get_line_info() : LineInfo()`

Returns a `LineInfo` structure representing the source location (file, line, column) of the call site where `get_line_info` is invoked.

`rtti::get_line_info(depth: int) : LineInfo()`

`rtti::get_total_functions(context: Context) : int()`

Returns the total number of registered functions (`int`) in the given `Context`.

Arguments

- **context** : *Context* implicit

`rtti::get_total_variables(context: Context) : int()`

Returns the total number of global variables (`int`) in the given `Context`.

Arguments

- **context** : *Context* implicit

`rtti::get_variable_info(context: any; index: int) : VarInfo()`

Returns the `VarInfo` pointer for a global variable at the given index in the `Context`, providing access to its name, type, and offset.

Arguments

- **context** : any
- **index** : int

`rtti::get_variable_value(varInfo: VarInfo) : RttiValue()`

Returns an `RttiValue` variant representing the current value of a global variable, looked up by `VarInfo` in the given `Context`.

Arguments

- **varInfo** : *VarInfo* implicit

`rtti::this_context() : Context&()`

Returns a pointer to the current `Context` in which the calling code is executing.

type_info

`rtti::type_info(cl: auto) : TypeInfo const?()`

Returns the `TypeInfo` object for the specified local variable or expression, resolved at compile time via the `[typeinfo(...)]` macro.

Arguments

- **cl** : auto

`rtti::type_info(vinfo: VarInfo) : TypeInfo const?()`

`rtti::type_info(vinfo: LocalVariableInfo) : TypeInfo const?()`

10.1.10 Program access

- *get_module (name: string) : Module?*
- *get_this_module (program: smart_ptr<Program>) : Module?*
- *program_for_each_module (program: smart_ptr<Program>; block: block<(Module?):void>)*
- *program_for_each_registered_module (block: block<(Module?):void>)*

`rtti::get_module(name: string) : Module?()`

Returns a `Module` pointer looked up by module name `string`, or null if no such module is registered.

Arguments

- **name** : string implicit

`rtti::get_this_module(program: smart_ptr<Program>) : Module?()`

Returns the `Module` pointer for the module currently being compiled or inferred, retrieved from the `Program`.

Arguments

- **program** : smart_ptr< *Program*> implicit

`rtti::program_for_each_module`(*program: smart_ptr<Program>; block: block<(Module?):void>*)

Iterates through all modules referenced by the given `Program` (including transitive dependencies), yielding a `Module` pointer for each.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **block** : block<(*Module*?):void> implicit

`rtti::program_for_each_registered_module`(*block: block<(Module?):void>*)

Iterates through all modules registered in the daslang runtime (globally, not per-program), yielding a `Module` pointer for each.

Arguments

- **block** : block<(*Module*?):void> implicit

10.1.11 Module access

- *module_for_each_annotation (module: Module?; block: block<(Annotation):void>)*
- *module_for_each_dependency (module: Module?; block: block<(Module?;bool):void>)*
- *module_for_each_enumeration (module: Module?; block: block<(EnumInfo):void>)*
- *module_for_each_function (module: Module?; block: block<(FuncInfo):void>)*
- *module_for_each_generic (module: Module?; block: block<(FuncInfo):void>)*
- *module_for_each_global (module: Module?; block: block<(VarInfo):void>)*
- *module_for_each_structure (module: Module?; block: block<(StructInfo):void>)*

`rtti::`**`module_for_each_annotation`**(*module: Module?; block: block<(Annotation):void>*)

Iterates through each annotation (handled type) in the given `Module`, yielding an `Annotation` pointer for each registered annotation.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*Annotation*):void> implicit

`rtti::`**`module_for_each_dependency`**(*module: Module?; block: block<(Module?;bool):void>*)

Iterates through each module dependency of the given `Module`, yielding the dependent `Module` pointer for each required module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*Module*?;bool):void> implicit

`rtti::`**`module_for_each_enumeration`**(*module: Module?; block: block<(EnumInfo):void>*)

Iterates through each enumeration declared in the given `Module`, yielding an `EnumInfo` pointer for each enum.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*EnumInfo*):void> implicit

`rtti::`**`module_for_each_function`**(*module: Module?; block: block<(FuncInfo):void>*)

Iterates through each function declared in the given `Module`, yielding a `FuncInfo` pointer for each function.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*FuncInfo*):void> implicit

`rtti::`**`module_for_each_generic`**(*module: Module?; block: block<(FuncInfo):void>*)

Iterates through each generic (template) function declared in the given `Module`, yielding a `FuncInfo` pointer for each generic.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*FuncInfo*):void> implicit

rtti::**module_for_each_global**(*module: Module?; block: block<(VarInfo):void>*)

Iterates through each global variable declared in the given `Module`, yielding a `VarInfo` pointer for each variable.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*VarInfo*):void> implicit

rtti::**module_for_each_structure**(*module: Module?; block: block<(StructInfo):void>*)

Iterates through each structure declaration in the given `Module`, yielding a `StructInfo` pointer for each struct.

Arguments

- **module** : *Module*? implicit
- **block** : block<(*StructInfo*):void> implicit

10.1.12 Annotation access

- *add_annotation_argument (annotation: AnnotationArgumentList; name: string) : int*
- *get_annotation_argument_value (info: AnnotationArgument) : RttiValue*

rtti::**add_annotation_argument(annotation: AnnotationArgumentList; name: string) : int**()

Appends an annotation argument (name-value pair) to the given `AnnotationArgumentList`, used when constructing annotations programmatically.

Arguments

- **annotation** : *AnnotationArgumentList* implicit
- **name** : string implicit

rtti::**get_annotation_argument_value(info: AnnotationArgument) : RttiValue**()

Returns an `RttiValue` variant representing the value of a specific named argument from an `AnnotationArgumentList`.

Arguments

- **info** : *AnnotationArgument* implicit

10.1.13 Compilation and simulation

- *compile (module_name: string; codeText: string; codeOfPolicies: CodeOfPolicies; exportAll: bool; block: block<(bool;smart_ptr<Program>;das_string):void>)*
- *compile (module_name: string; codeText: string; codeOfPolicies: CodeOfPolicies; block: block<(bool;smart_ptr<Program>;das_string):void>)*
- *compile_file (module_name: string; fileAccess: smart_ptr<FileAccess>; moduleGroup: ModuleGroup?; codeOfPolicies: CodeOfPolicies; block: block<(bool;smart_ptr<Program>;das_string):void>)*
- *for_each_expected_error (program: smart_ptr<Program>; block: block<(CompilationError;int):void>)*
- *for_each_require_declaration (program: smart_ptr<Program>; block: block<(Module?;string#;string#;bool;LineInfo):void>)*
- *simulate (program: smart_ptr<Program> const&; block: block<(bool;smart_ptr<Context>;das_string):void>)*

compile

rtti::**compile**(*module_name: string; codeText: string; codeOfPolicies: CodeOfPolicies; exportAll: bool; block: block<(bool;smart_ptr<Program>;das_string):void>*)

Compiles a daslang program from a source code string using the provided `FileAccess` and `ModuleGroup`, returning a `ProgramPtr` (null on failure).

Arguments

- **module_name** : string implicit
- **codeText** : string implicit
- **codeOfPolicies** : *CodeOfPolicies* implicit
- **exportAll** : bool
- **block** : block<(bool;smart_ptr< *Program*>; *das_string*):void> implicit

rtti::**compile**(*module_name: string; codeText: string; codeOfPolicies: CodeOfPolicies; block: block<(bool;smart_ptr<Program>;das_string):void>*)

---

rtti::**compile_file**(*module_name: string; fileAccess: smart_ptr<FileAccess>; moduleGroup: ModuleGroup?; codeOfPolicies: CodeOfPolicies; block: block<(bool;smart_ptr<Program>;das_string):void>*)

Compiles a daslang program from a file registered in the given `FileAccess` object, returning a `ProgramPtr` (null on failure).

Arguments

- **module_name** : string implicit
- **fileAccess** : smart_ptr< *FileAccess*> implicit
- **moduleGroup** : *ModuleGroup*? implicit
- **codeOfPolicies** : *CodeOfPolicies* implicit
- **block** : block<(bool;smart_ptr< *Program*>; *das_string*):void> implicit

rtti::**for_each_expected_error**(*program: smart_ptr<Program>; block: block<(CompilationError;int):void>*)

Iterates through each expected compilation error declared in the `Program` (via `expect`), yielding the error code for each.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **block** : block<(*CompilationError*;int):void> implicit

rtti::**for_each_require_declaration**(*program: smart_ptr<Program>; block: block<(Module?;string#;string#;bool;LineInfo):void>*)

Iterates through each `require` declaration of the compiled `Program`, yielding the module name, public/private flag, and source `LineInfo`.

Arguments

- **program** : smart_ptr< *Program*> implicit

- **block** : block<(*Module*?;string#;string#;bool; *LineInfo*&):void> implicit

`rtti::`**`simulate`**`(`*program: smart_ptr<Program> const&; block: block<(bool;smart_ptr<Context>;das_string):void>*`)`

Simulates (links and initializes) a compiled `Program`, returning a `Context` pointer ready for function execution, or null on failure.

Arguments

- **program** : smart_ptr< *Program*>& implicit
- **block** : block<(bool;smart_ptr< *Context*>; *das_string*):void> implicit

10.1.14 File access

- *add_file_access_root (access: smart_ptr<FileAccess>; mod: string; path: string) : bool*
- *make_file_access (project: string) : smart_ptr<FileAccess>*
- *set_file_source (access: smart_ptr<FileAccess>; fileName: string; text: string) : bool*

`rtti::`**`add_file_access_root(access: smart_ptr<FileAccess>; mod: string; path: string) : bool`**`()`

Adds an extra root directory (search path) to the given `FileAccess` object, expanding where `require` resolves files from.

Arguments

- **access** : smart_ptr< *FileAccess*> implicit
- **mod** : string implicit
- **path** : string implicit

`rtti::`**`make_file_access(project: string) : smart_ptr<FileAccess>`**`()`

Creates and returns a new `FileAccessPtr` (`smart_ptr<FileAccess>`) initialized as a default file-system-backed project.

Arguments

- **project** : string implicit

`rtti::`**`set_file_source(access: smart_ptr<FileAccess>; fileName: string; text: string) : bool`**`()`

Registers a source code `string` for the given file name inside the `FileAccess` object, allowing in-memory compilation without disk files.

Arguments

- **access** : smart_ptr< *FileAccess*> implicit
- **fileName** : string implicit
- **text** : string implicit

10.1.15 Structure access

- *basic_struct_for_each_field (annotation: BasicStructureAnnotation; block: block<(string;string;TypeInfo;uint):void>)*
- *basic_struct_for_each_parent (annotation: BasicStructureAnnotation; block: block<(Annotation?):void>)*
- *rtti_builtin_structure_for_each_annotation (struct: StructInfo; block: block<():void>)*
- *structure_for_each_annotation (st: StructInfo; subexpr: block<(ann:Annotation;args:AnnotationArguments):void>) : auto*

`rtti::`**`basic_struct_for_each_field`**`(`*annotation: BasicStructureAnnotation; block: block<(string;string;TypeInfo;uint):void>*`)`

Iterates through each field of a `BasicStructureAnnotation`, yielding the field name, C++ name, `TypeInfo`, and byte offset for each field.

Arguments

- **annotation** : *BasicStructureAnnotation* implicit
- **block** : block<(string;string; *TypeInfo*;uint):void> implicit

`rtti::`**`basic_struct_for_each_parent`**`(`*annotation: BasicStructureAnnotation; block: block<(Annotation?):void>*`)`

Iterates through each parent (base class) of a `BasicStructureAnnotation`, yielding the parent `TypeInfo` for each ancestor.

Arguments

- **annotation** : *BasicStructureAnnotation* implicit
- **block** : block<(*Annotation*?):void> implicit

`rtti::`**`rtti_builtin_structure_for_each_annotation`**`(`*struct: StructInfo; block: block<():void>*`)`

Iterates through each annotation attached to a `StructInfo`, yielding the annotation name and its `AnnotationArgumentList` for each.

Arguments

- **struct** : *StructInfo* implicit
- **block** : block<void> implicit

`rtti::`**`structure_for_each_annotation(st: StructInfo; subexpr: block<(ann:Annotation;args:AnnotationArgume`**

Iterates through each annotation attached to a `StructInfo`, yielding the annotation name and `AnnotationArgumentList` — an alias of `rtti_builtin_structure_for_each_annotation`.

Arguments

- **st** : *StructInfo*
- **subexpr** : block<(ann: *Annotation*;args: *AnnotationArguments*):void>

10.1.16 Data walking and printing

- *describe (lineinfo: LineInfo; fully: bool = false) : string*
- *describe (type: TypeInfo const?) : string*
- *get_mangled_name (type: TypeInfo const?) : string*
- *sprint_data (data: float4; type: TypeInfo const?; flags: bitfield) : string*
- *sprint_data (data: void?; type: TypeInfo const?; flags: bitfield) : string*

describe

`rtti::`**`describe(lineinfo: LineInfo; fully: bool = false) : string`**`()`

Returns a human-readable `string` description of an RTTI object (`TypeInfo`, `VarInfo`, `FuncInfo`, etc.), useful for logging and debug output.

Arguments

- **lineinfo** : *LineInfo* implicit
- **fully** : bool

`rtti::`**`describe(type: TypeInfo const?) : string`**`()`

`rtti::`**`get_mangled_name(type: TypeInfo const?) : string`**`()`

Returns the full mangled name `string` for the given `FuncInfo`, encoding its module, name, and argument types.

Arguments

- **type** : *TypeInfo*? implicit

sprint_data

`rtti::`**`sprint_data(data: float4; type: TypeInfo const?; flags: bitfield) : string`**`()`

Returns a `string` representation of a value given its data pointer and `TypeInfo`, similar to `debug` or `print` but capturing output as a string.

Arguments

- **data** : float4
- **type** : *TypeInfo*? implicit
- **flags** : bitfield<>

`rtti::`**`sprint_data(data: void?; type: TypeInfo const?; flags: bitfield) : string`**`()`

10.1.17 Function and mangled name hash

- *get_function_address (MNH: uint64; at: Context) : uint64*
- *get_function_by_mangled_name_hash (src: uint64; context: Context) : function<():void>*
- *get_function_by_mangled_name_hash (src: uint64) : function<():void>*
- *get_function_mangled_name_hash (src: function<():void>) : uint64*

rtti::**get_function_address(MNH: uint64; at: Context) : uint64**()

Returns a `SimFunction` pointer looked up by mangled name hash in the given `Context`, or null if not found.

Arguments

- **MNH** : uint64
- **at** : *Context* implicit

get_function_by_mangled_name_hash

rtti::**get_function_by_mangled_name_hash(src: uint64; context: Context) : function<():void>**()

Returns a `function`<> lambda value looked up by its mangled name hash in the given `Context`.

Arguments

- **src** : uint64
- **context** : *Context* implicit

rtti::**get_function_by_mangled_name_hash(src: uint64) : function<():void>**()

---

rtti::**get_function_mangled_name_hash(src: function<():void>) : uint64**()

Returns the `uint64` mangled name hash for the given `function`<> value, which uniquely identifies the function in its `Context`.

Arguments

- **src** : function<void>

10.1.18 Context and mutex locking

- *lock_context (lock_context: Context; block: block<():void>)*
- *lock_mutex (mutex: recursive_mutex; block: block<():void>)*
- *lock_this_context (block: block<():void>)*

rtti::**lock_context**(*lock_context: Context; block: block<():void>*)

Acquires a recursive lock on the given `Context` and executes a block, ensuring thread-safe access to context data within the scope.

Arguments

- **lock_context** : *Context* implicit
- **block** : block<void> implicit

`rtti::`**`lock_mutex`**`(`*mutex: recursive_mutex; block: block<():void>*`)`

Acquires a recursive lock on the given `recursive_mutex` and executes a block, releasing the lock when the block exits.

Arguments

- **mutex** : *recursive_mutex* implicit
- **block** : block<void> implicit

`rtti::`**`lock_this_context`**`(`*block: block<():void>*`)`

Acquires a recursive lock on the current `Context` and executes a block, ensuring thread-safe access within the scope.

Arguments

- **block** : block<void> implicit

10.1.19 Runtime data access

- *get_table_key_index (table: void?; key: any; baseType: Type; valueTypeSize: int) : int*

`rtti::`**`get_table_key_index(table: void?; key: any; baseType: Type; valueTypeSize: int) : int`**`()`

Returns the internal slot index (`int`) for the given key within a `table` value, or `-1` if the key is not present.

Arguments

- **table** : void? implicit
- **key** : any
- **baseType** : *Type*
- **valueTypeSize** : int

10.1.20 Tuple and variant access

- *get_tuple_field_offset (type: TypeInfo?; index: int) : int*
- *get_variant_field_offset (type: TypeInfo?; index: int) : int*

`rtti::`**`get_tuple_field_offset(type: TypeInfo?; index: int) : int`**`()`

Returns the byte offset (`int`) of a field at the given index within a tuple type described by `TypeInfo`.

Arguments

- **type** : *TypeInfo*? implicit
- **index** : int

`rtti::`**`get_variant_field_offset(type: TypeInfo?; index: int) : int`**`()`

Returns the byte offset (`int`) of a field at the given index within a variant type described by `TypeInfo`.

Arguments

- **type** : *TypeInfo*? implicit
- **index** : int

10.1.21 Iteration

- *each (info: FuncInfo const implicit ==const) : iterator<VarInfo const&>*
- *each (info: FuncInfo implicit ==const) : iterator<VarInfo&>*
- *each (info: StructInfo const implicit ==const) : iterator<VarInfo const&>*
- *each (info: StructInfo implicit ==const) : iterator<VarInfo&>*
- *each (info: EnumInfo const implicit ==const) : iterator<EnumValueInfo const&>*
- *each (info: EnumInfo implicit ==const) : iterator<EnumValueInfo&>*

each

`rtti::`**`each(info: FuncInfo const implicit ==const) : iterator<VarInfo const&>`**`()`

Iterates through each element of an RTTI container (e.g., `AnnotationArguments`, `AnnotationArgumentList`, `AnnotationList`), yielding individual entries.

Arguments

- **info** : *FuncInfo* implicit!

`rtti::`**`each(info: FuncInfo implicit ==const) : iterator<VarInfo&>`**`()`

`rtti::`**`each(info: StructInfo const implicit ==const) : iterator<VarInfo const&>`**`()`

`rtti::`**`each(info: StructInfo implicit ==const) : iterator<VarInfo&>`**`()`

`rtti::`**`each(info: EnumInfo const implicit ==const) : iterator<EnumValueInfo const&>`**`()`

`rtti::`**`each(info: EnumInfo implicit ==const) : iterator<EnumValueInfo&>`**`()`

10.2 AST manipulation library

The AST module provides access to the abstract syntax tree representation of daslang programs. It defines node types for all language constructs (expressions, statements, types, functions, structures, enumerations, etc.), visitors for tree traversal, and utilities for AST construction and manipulation. This module is the foundation for writing macros, code generators, and source-level program transformations.

All functions and symbols are in “ast” module, use require to get access to it.

```
require ast
```

10.2.1 Type aliases

`ast::`**`bitfield TypeDeclFlags`**

properties of the *TypeDecl* object.

Fields

- **ref** (0x1) - The type is a reference type.
- **constant** (0x2) - The type is a constant type.

- **temporary** (0x4) - The type is a temporary type.
- **_implicit** (0x8) - The type is an implicit type.
- **removeRef** (0x10) - Remove the reference flag.
- **removeConstant** (0x20) - Remove the constant flag.
- **removeDim** (0x40) - Remove the dimension flag.
- **removeTemporary** (0x80) - Remove the temporary flag.
- **explicitConst** (0x100) - The type is an explicit constant type.
- **aotAlias** (0x200) - The type is an AOT alias.
- **smartPtr** (0x400) - The type is a smart pointer type.
- **smartPtrNative** (0x800) - The type is a native smart pointer type (smart_ptr_raw).
- **isExplicit** (0x1000) - The type is explicit.
- **isNativeDim** (0x2000) - The type is a native dimension.
- **isTag** (0x4000) - The type is a reification tag.
- **explicitRef** (0x8000) - The type is an explicit reference.
- **isPrivateAlias** (0x10000) - The type is a private alias.
- **autoToAlias** (0x20000) - The type is an auto-to-alias.

`ast::`**`bitfield FieldDeclarationFlags`**

properties of the *FieldDeclaration* object.

Fields

- **moveSemantics** (0x1) - The field is initialized using move semantics.
- **parentType** (0x2) - Which parent type this field belongs to.
- **capturedConstant** (0x4) - This field is a captured constant (via lambda or generator).
- **generated** (0x8) - This field is compiler-generated.
- **capturedRef** (0x10) - This field is a captured reference (via lambda or generator).
- **doNotDelete** (0x20) - Has @do_not_delete attribute.
- **privateField** (0x40) - This field is private.
- **_sealed** (0x80) - The field is sealed. It cannot be overridden in derived types.
- **implemented** (0x100) - Already implemented.
- **classMethod** (0x200) - This field is a class method.

`ast::`**`bitfield StructureFlags`**

properties of the *Structure* object.

Fields

- **isClass** (0x1) - The structure is a class.
- **genCtor** (0x2) - Generate constructor.
- **cppLayout** (0x4) - C++ data layout.

- **cppLayoutNotPod** (0x8) - C++ layout not POD type, i.e. has alignment to accommodate for inheritance.
- **generated** (0x10) - This structure is compiler-generated.
- **persistent** (0x20) - This structure is using persistent heap (C++ heap).
- **isLambda** (0x40) - This structure is a lambda.
- **privateStructure** (0x80) - This structure is private.
- **macroInterface** (0x100) - This structure is a macro interface.
- **_sealed** (0x200) - This structure is sealed. It cannot be inherited.
- **skipLockCheck** (0x400) - Skip lock check.
- **circular** (0x800) - This structure has circular references (and is invalid).
- **_generator** (0x1000) - This structure is a generator.
- **hasStaticMembers** (0x2000) - This structure has static members.
- **hasStaticFunctions** (0x4000) - This structure has static functions.
- **hasInitFields** (0x8000) - This structure has initialized fields.
- **safeWhenUninitialized** (0x10000) - This structure is safe when uninitialized.
- **isTemplate** (0x20000) - This structure is a template.
- **hasDefaultInitializer** (0x40000) - This structure has a default initializer.
- **noGenCtor** (0x80000) - This structure does not generate a default constructor.

`ast::`**`bitfield ExprGenFlags`**

generation (genFlags) properties of the *Expression* object.

Fields

- **alwaysSafe** (0x1) - Expression is always safe.
- **generated** (0x2) - Expression is compiler-generated.
- **userSaidItsSafe** (0x4) - Expression is marked as safe explicitly.

`ast::`**`bitfield ExprLetFlags`**

properties of the *ExprLet* object.

Fields

- **inScope** (0x1) - It's 'let inscope' expression.
- **hasEarlyOut** (0x2) - It's 'let hasEarlyOut' expression.
- **itTupleExpansion** (0x4) - It's 'let itTupleExpansion' expression.

`ast::`**`bitfield ExprFlags`**

properties of the *Expression* object.

Fields

- **constexpression** (0x1) - Expression is a constant expression.
- **noSideEffects** (0x2) - Expression has no side effects.

- **noNativeSideEffects** (0x4) - Expression has no native side effects, i.e. expression itself has no sideeffects.
- **isForLoopSource** (0x8) - Expression is a for loop source.
- **isCallArgument** (0x10) - Expression is a call argument.

`ast::bitfield ExprPrintFlags`

printing properties of the *Expression* object.

Fields

- **topLevel** (0x1) - Its a top level expression.
- **argLevel** (0x2) - Its an argument level expression.
- **bottomLevel** (0x4) - Its a bottom level expression - no sub-expressions or nesting.

`ast::bitfield FunctionFlags`

properties of the *Function* object.

Fields

- **builtIn** (0x1) - Function is built-in.
- **policyBased** (0x2) - Function is policy-based.
- **callBased** (0x4) - Function is call-based.
- **interopFn** (0x8) - Function is interop function.
- **hasReturn** (0x10) - Function has a return value.
- **copyOnReturn** (0x20) - Function copies return value.
- **moveOnReturn** (0x40) - Function moves return value.
- **exports** (0x80) - Its an exported function.
- **init** (0x100) - Its an init function.
- **addr** (0x200) - Function has address requested.
- **used** (0x400) - Function is used.
- **fastCall** (0x800) - Function is fast call.
- **knownSideEffects** (0x1000) - Function has known side effects (user defined).
- **hasToRunAtCompileTime** (0x2000) - Function has to run at compile time.
- **unsafeOperation** (0x4000) - Function is unsafe operation.
- **unsafeDeref** (0x8000) - All dereferences in the function will be simulated without safety checks.
- **hasMakeBlock** (0x10000) - Function has 'make block' operation.
- **aotNeedPrologue** (0x20000) - Function needs AOT prologue.
- **noAot** (0x40000) - Function is not AOT.
- **aotHybrid** (0x80000) - Function is AOT hybrid, i.e. can be called from both AOT and interpreted code. Call in never hardcoded.
- **aotTemplate** (0x100000) - Function is AOT template, i.e. instantiated from template at C++ compile time.
- **generated** (0x200000) - Function is compiler-generated.

- **privateFunction** (0x400000) - Function is private.
- **_generator** (0x800000) - Function is a generator.
- **_lambda** (0x1000000) - Function is a lambda.
- **firstArgReturnType** (0x2000000) - First argument type is return type.
- **noPointerCast** (0x4000000) - Function has no pointer cast.
- **isClassMethod** (0x8000000) - Function is a class method.
- **isTypeConstructor** (0x10000000) - Function is a type constructor.
- **shutdown** (0x20000000) - Function is a shutdown function.
- **anyTemplate** (0x40000000) - Function is any template.
- **macroInit** (0x80000000) - Function is macro init.

`ast::bitfield MoreFunctionFlags`

additional properties of the *Function* object.

Fields

- **macroFunction** (0x1) - Function is a macro function.
- **needStringCast** (0x2) - Converts das string arguments to C++ `char *`. Empty string, which is null in das, is converted to "".
- **aotHashDeppendsOnArguments** (0x4) - Function hash depends on arguments.
- **lateInit** (0x8) - Function is late initialized.
- **requestJit** (0x10) - Function is requested to be JIT compiled.
- **unsafeOutsideOfFor** (0x20) - Function is unsafe outside of for loop sources.
- **skipLockCheck** (0x40) - Skip lock check for this function.
- **safeImplicit** (0x80) - Function is safe for implicit calls. Otherwise temp values are to be specialized for in the generic.
- **deprecated** (0x100) - Function is deprecated.
- **aliasCMRES** (0x200) - Function aliases CMRES (Copy or Move return result).
- **neverAliasCMRES** (0x400) - Function never aliases CMRES.
- **addressTaken** (0x800) - Function address is taken.
- **propertyFunction** (0x1000) - Function is a property function.
- **pinvoke** (0x2000) - Function is a P/Invoke function, i.e. cross-context call.
- **jitOnly** (0x4000) - Function is JIT only.
- **isStaticClassMethod** (0x8000) - Function is a static class method.
- **requestNoJit** (0x10000) - Function is requested to not be JIT compiled.
- **jitContextAndLineInfo** (0x20000) - Function requires JIT context and line info.
- **nodiscard** (0x40000) - Discarding the return value of the function is unsafe.
- **captureString** (0x80000) - Function captures string arguments.
- **callCaptureString** (0x100000) - Function calls capture string arguments.
- **hasStringBuilder** (0x200000) - Function has a string builder.

- **recursive** (0x400000) - Function is recursive.
- **isTemplate** (0x800000) - Function is a template function.
- **unsafeWhenNotCloneArray** (0x1000000) - Function is unsafe, when its not used to clone arrays.
- **stub** (0x2000000) - This flag is a stub.
- **lateShutdown** (0x4000000) - Function will shutdown after all other shutdonws are done.
- **hasTryRecover** (0x8000000) - Function has tryrecover blocks.
- **hasUnsafe** (0x10000000) - Function has unsafe operations made by user.
- **isConstClassMethod** (0x20000000) - Function is a const class method.

`ast::bitfield FunctionSideEffectFlags`

side-effect properties of the *Function* object.

Fields

- **_unsafe** (0x1) - Function is unsafe.
- **userScenario** (0x2) - User specified [sideeffects] annotation to indicate side effects.
- **modifyExternal** (0x4) - Function may modify external state.
- **modifyArgument** (0x8) - Function may modify argument values.
- **accessGlobal** (0x10) - Function may access global state (variables and such).
- **invoke** (0x20) - Function is using ‘invoke’, so we don’t know any additional side effects.

`ast::bitfield VariableFlags`

properties of the *Variable* object.

Fields

- **init_via_move** (0x1) - Variable is initialized via move <-
- **init_via_clone** (0x2) - Variable is initialized via clone :=
- **used** (0x4) - Variable is used
- **aliasCMRES** (0x8) - Variable is an alias for CMRES return value
- **marked_used** (0x10) - Variable is marked as used (to suppress unused warnings)
- **global_shared** (0x20) - Variable is a global shared variable
- **do_not_delete** (0x40) - @do_not_delete annotation on the variable
- **generated** (0x80) - Variable is generated by the compiler
- **capture_as_ref** (0x100) - Variable is captured by reference in a closure
- **can_shadow** (0x200) - Variable can shadow another variable in an inner scope
- **private_variable** (0x400) - Variable is private to the class/struct
- **tag** (0x800) - Variable is a reification tag
- **global** (0x1000) - Variable is a global variable
- **inScope** (0x2000) - Variable is ‘let inscope’, i.e. there is a coresponding ‘delete’ in the ‘finally’ section of the block

- **no_capture** (0x4000) - This variable will not be captured in lambda (think 'self').
- **early_out** (0x8000) - There is an early out from the scope where this variable is defined (via return and otherwise)
- **used_in_finally** (0x10000) - Variable is used in the finally block
- **static_class_member** (0x20000) - Variable is a static class member
- **bitfield_constant** (0x40000) - Variable is a bitfield constant
- **pod_delete** (0x80000) - This variable can be deleted as POD
- **pod_delete_gen** (0x100000) - POD delete has been generated for this variable
- **single_return_via_move** (0x200000) - This variable is returned via move in a function with only one return path

`ast::bitfield VariableAccessFlags`

access properties of the *Variable* object.

Fields

- **access_extern** (0x1) - Variable is Function or block argument.
- **access_get** (0x2) - Variable is accessed via get (read of some kind).
- **access_ref** (0x4) - Variable is accessed via ref (written to).
- **access_init** (0x8) - Variable is initialized.
- **access_pass** (0x10) - Variable is passed to a function, or invoke.
- **access_fold** (0x20) - Variable was folded aways (optimized out).

`ast::bitfield ExprBlockFlags`

properties of the *ExprBlock* object.

Fields

- **isClosure** (0x1) - Block is a closure, and not a regular expression list.
- **hasReturn** (0x2) - Block has a return statement.
- **copyOnReturn** (0x4) - When invoked, the block result is copied on return.
- **moveOnReturn** (0x8) - When invoked, the block result is moved on return.
- **inTheLoop** (0x10) - Block is inside a loop.
- **finallyBeforeBody** (0x20) - Finally is to be visited before the body.
- **finallyDisabled** (0x40) - Finally is disabled.
- **aotSkipMakeBlock** (0x80) - AOT is allowed to skip make block generation, and pass [&]() directly.
- **aotDoNotSkipAnnotationData** (0x100) - AOT should not skip annotation data even if make block is skipped.
- **isCollapseable** (0x200) - Block is eligible for collapse optimization.
- **needCollapse** (0x400) - Block needs to be collapsed.
- **hasMakeBlock** (0x800) - Block has make block operation.
- **hasEarlyOut** (0x1000) - Block has early out (break/continue/return).

- **forLoop** (0x2000) - Block is a for loop body.
- **hasExitByLabel** (0x4000) - Block has exit by label (goto outside).
- **isLambdaBlock** (0x8000) - Block is a lambda block.
- **isGeneratorBlock** (0x10000) - Block is a generator block.

`ast::`**`bitfield ExprAtFlags`**

properties of the *ExprAt* object.

Fields

- **r2v** (0x1) - Reference to value conversion is applied.
- **r2cr** (0x2) - Read to const reference is propagated.
- **write** (0x4) - The result is written to.
- **no_promotion** (0x8) - Promotion to operator is disabled, even if operator [] is overloaded.
- **under_clone** (0x10) - The expression is under a clone operation.

`ast::`**`bitfield ExprMakeLocalFlags`**

properties of the *ExprMakeLocal* object (*ExprMakeArray*, *ExprMakeStruct*, 'ExprMakeTuple', 'ExprMakeVariant').

Fields

- **useStackRef** (0x1) - Use stack reference, i.e. there is an address on the stack - where the reference is written to.
- **useCMRES** (0x2) - Result is returned via CMRES pointer. Usually this is 'return <- [[ExprMakeLocal]]'
- **doesNotNeedSp** (0x4) - Does not need stack pointer, usually due to being part of bigger initialization.
- **doesNotNeedInit** (0x8) - Does not need field initialization, usually due to being fully initialized via constructor.
- **initAllFields** (0x10) - Initialize all fields.
- **alwaysAlias** (0x20) - Always alias the result, so temp value is always allocated on the stack.

`ast::`**`bitfield ExprAscendFlags`**

properties of the *ExprAscend* object.

Fields

- **useStackRef** (0x1) - Use stack reference, i.e. there is an address on the stack - where the reference is written to.
- **needTypeInfo** (0x2) - Simulated node needs type information at runtime.
- **isMakeLambda** (0x4) - Is a lambda expression.

`ast::`**`bitfield ExprCastFlags`**

properties of the *ExprCast* object.

Fields

- **upcastCast** (0x1) - Upcast cast, i.e. casting from based class to derived class.
- **reinterpretCast** (0x2) - Reinterpret cast, i.e. casting between unrelated types (like pointer to integer)

`ast::bitfield ExprVarFlags`

properties of the *ExprVar* object.

Fields

- **local** (0x1) - Local variable.
- **argument** (0x2) - Function argument.
- **_block** (0x4) - Block argument
- **thisBlock** (0x8) - Argument of the most-nested block.
- **r2v** (0x10) - Reference to value conversion is applied.
- **r2cr** (0x20) - Read to const reference is propagated.
- **write** (0x40) - Written to.
- **under_clone** (0x80) - This is a foo := bar expression, and the variable is being cloned to.

`ast::bitfield ExprMakeStructFlags`

properties of the *ExprMakeStruct* object.

Fields

- **useInitializer** (0x1) - Use initializer, i.e. 'Foo(…)', and not 'Foo(uninitialized …)'.
- **isNewHandle** (0x2) - Its 'new Foo(…)'.
- **usedInitializer** (0x4) - 'useInitializer' was used optimized out.
- **nativeClassInitializer** (0x8) - Generated class initializer.
- **isNewClass** (0x10) - Its 'new ClassName(…)'.
- **forceClass** (0x20) - Its declared via 'class'syntax, so using it for regular types will fail.
- **forceStruct** (0x40) - Its declared via 'struct' syntax, so using it for regular types will fail.
- **forceVariant** (0x80) - Its declared via 'variant' syntax, so using it for regular types will fail.
- **forceTuple** (0x100) - Its declared via 'tuple' syntax, so using it for regular types will fail.
- **alwaysUseInitializer** (0x200) - Always use initializer, even for default construction.
- **ignoreVisCheck** (0x400) - Ignores visibility check between modules.
- **canShadowBlock** (0x800) - 'where' section argument can shadow other variables. This is for nested comprehensions and such.

`ast::bitfield MakeFieldDeclFlags`

Properties of the *MakeFieldDecl* object.

Fields

- **moveSemantics** (0x1) - Initialized with move semantics, <-
- **cloneSemantics** (0x2) - Initialized with clone semantics, :=

`ast::bitfield ExprFieldDerefFlags`

dereferencing properties of the *ExprField* object.

Fields

- **unsafeDeref** (0x1) - Dereference without safety checking.

- **ignoreCaptureConst** (0x2) - Ignore capture const, i.e. was captured as constant - but used as mutable.

`ast::bitfield ExprFieldFieldFlags`

field properties of the *ExprField* object.

Fields

- **r2v** (0x1) - Reference to value conversion is applied.
- **r2cr** (0x2) - Read to const reference is propagated.
- **write** (0x4) - This is part of a write operation, and a field or part of it is being assigned to.
- **no_promotion** (0x8) - No promotion to property, even if available.
- **under_clone** (0x10) - Under clone, i.e. 'Foo.bar := …'

`ast::bitfield ExprSwizzleFieldFlags`

properties of the *ExprSwizzle* object.

Fields

- **r2v** (0x1) - Reference to value conversion is applied.
- **r2cr** (0x2) - Read to const reference is propagated.
- **write** (0x4) - This is part of a write operation, and a field or part of it is being assigned to.

`ast::bitfield ExprYieldFlags`

properties of the *ExprYield* object.

Fields

- **moveSemantics** (0x1) - Its 'yield <- …'.
- **skipLockCheck** (0x2) - Skip lock checks.

`ast::bitfield ExprReturnFlags`

properties of the *ExprReturn* object.

Fields

- **moveSemantics** (0x1) - Its 'return <- …'.
- **returnReference** (0x2) - Return a reference. Function result is a reference.
- **returnInBlock** (0x4) - Return in block, not in function.
- **takeOverRightStack** (0x8) - Take over right stack, i.e its 'return [MakeLocal]' and temp stack value is allocated by return expression.
- **returnCallCMRES** (0x10) - Return call CMRES, i.e. 'return call(…)'.
- **returnCMRES** (0x20) - Return CMRES, i.e. 'return [MakeLocal]' or 'return [CmresVariable]'
- **fromYield** (0x40) - From yield.
- **fromComprehension** (0x80) - From comprehension.
- **skipLockCheck** (0x100) - Skip lock checks.

ast::**bitfield ExprMakeBlockFlags**

properties of the *ExprMakeBlock* object.

Fields

- **isLambda** (0x1) - Is lambda, i.e. @(. . .) { . . . }
- **isLocalFunction** (0x2) - Is a local function, i.e. @@(. . .) { . . . }

ast::**bitfield CopyFlags**

properties of the *ExprCopy* object.

Fields

- **allowCopyTemp** (0x1) - This copy is allowed to copy a temporary value.
- **takeOverRightStack** (0x2) - Its 'foo = [MakeLocal]' and temp stack value is allocated by copy expression.
- **allowConstantLValue** (0x4) - Promote to clone, i.e. this is 'foo := bar' and not 'foo = bar'

ast::**bitfield MoveFlags**

Properties of the *ExprMove* object.

Fields

- **skipLockCheck** (0x1) - Skip lock checks.
- **takeOverRightStack** (0x2) - Its 'foo <- [MakeLocal]' and temp stack value is allocated by move expression.
- **allowConstantLValue** (0x4) - Move is allowed for constant lvalue, for example x <- 5
- **podDelete** (0x8) - Move is a POD delete.

ast::**bitfield IfFlags**

properties of the *ExprIf* object.

Fields

- **isStatic** (0x1) - This is a 'static_if' or 'static_elif' expression.
- **doNotFold** (0x2) - Do not fold this 'if' expression during compilation.

ast::**bitfield StringBuilderFlags**

properties of the *ExprStringBuilder* object.

Fields

- **isTempString** (0x1) - String builder produces a temporary string.

ast::**ExpressionPtr = smart_ptr<Expression>**

Smart pointer to an *Expression* object. The fundamental handle type for all AST expression nodes.

ast::**ProgramPtr = smart_ptr<Program>**

Smart pointer to a *Program* object. Represents the root of a compiled daslang program, containing all modules, functions, and structures.

ast::**TypeDeclPtr = smart_ptr<TypeDecl>**

Smart pointer to a *TypeDecl* object. The fundamental handle type for all type declarations in the AST type system.

`ast::VectorTypeDeclPtr = dasvector`smart_ptr`TypeDecl`

Smart pointer to a `das::vector<ExpressionPtr>`. Represents an ordered collection of type declarations, typically used for function argument lists and tuple fields.

`ast::EnumerationPtr = smart_ptr<Enumeration>`

Smart pointer to an *Enumeration* object. Used for creating and manipulating enumeration declarations in the AST.

`ast::StructurePtr = smart_ptr<Structure>`

Smart pointer to a *Structure* object. Used for creating and manipulating structure declarations in the AST.

`ast::FunctionPtr = smart_ptr<Function>`

Smart pointer to a *Function* object. Used for creating and manipulating function declarations in the AST.

`ast::VariablePtr = smart_ptr<Variable>`

Smart pointer to a *Variable* object. Used for creating and manipulating variable declarations in the AST.

`ast::MakeFieldDeclPtr = smart_ptr<MakeFieldDecl>`

Smart pointer to a *MakeFieldDecl* object. Represents a single field initializer in structure or variant construction expressions.

`ast::ExprMakeBlockPtr = smart_ptr<ExprMakeBlock>`

Smart pointer to an *ExprMakeBlock* expression. Wraps block or lambda creation expressions in the AST.

`ast::FunctionAnnotationPtr = smart_ptr<FunctionAnnotation>`

Smart pointer to a *FunctionAnnotation* object. Used for registering and managing function annotation macros.

`ast::StructureAnnotationPtr = smart_ptr<StructureAnnotation>`

Smart pointer to a *StructureAnnotation* object. Used for registering and managing structure annotation macros.

`ast::EnumerationAnnotationPtr = smart_ptr<EnumerationAnnotation>`

Smart pointer to an *EnumerationAnnotation* object. Used for registering and managing enumeration annotation macros.

`ast::PassMacroPtr = smart_ptr<PassMacro>`

Smart pointer to a *PassMacro* object. Used for registering and managing custom inference pass macros.

`ast::VariantMacroPtr = smart_ptr<VariantMacro>`

Smart pointer to a *VariantMacro* object. Used for registering and managing custom variant dispatch macros.

`ast::ReaderMacroPtr = smart_ptr<ReaderMacro>`

Smart pointer to a *ReaderMacro* object. Used for registering and managing custom reader (parsing) macros.

`ast::CommentReaderPtr = smart_ptr<CommentReader>`

Smart pointer to a *CommentReader* object. Used for registering and managing custom comment parsing macros.

`ast::CallMacroPtr = smart_ptr<CallMacro>`

Smart pointer to a *CallMacro* object. Used for registering and managing custom call-like expression macros.

`ast::TypeInfoMacroPtr = smart_ptr<TypeInfoMacro>`

Smart pointer to a *TypeInfoMacro* object. Used for registering and managing custom `typeinfo` trait macros.

`ast::ForLoopMacroPtr = smart_ptr<ForLoopMacro>`

Smart pointer to a *ForLoopMacro* object. Used for registering and managing custom for-loop macros.

`ast::CaptureMacroPtr = smart_ptr<CaptureMacro>`

Smart pointer to a *CaptureMacro* object. Used for registering and managing custom lambda capture macros.

`ast::TypeMacroPtr = smart_ptr<TypeMacro>`

Smart pointer to a *TypeMacro* object. Used for registering and managing custom type declaration macros.

`ast::SimulateMacroPtr = smart_ptr<SimulateMacro>`

Smart pointer to a *SimulateMacro* object. Used for registering and managing macros that hook into the simulation phase.

10.2.2 Enumerations

`ast::CaptureMode`

Enumeration with lambda variables capture modes.

Values

- **capture_any** = 0 - Unspecified capture mode (will try copy, then reference - and ask for unsafe).
- **capture_by_copy** = 1 - Value is copied.
- **capture_by_reference** = 2 - Reference to the original value is captured (this one is unsafe)
- **capture_by_clone** = 3 - Value is cloned.
- **capture_by_move** = 4 - Value is moved.

`ast::SideEffects`

Enumeration with all possible side effects of expression or function.

Values

- **none** = 0 - No side effects.
- **unsafe** = 1 - Function is unsafe.
- **userScenario** = 2 - [sideeffects] annotation to indicate side effects.
- **modifyExternal** = 4 - Function may modify external state.
- **accessExternal** = 4 - Access to external state.
- **modifyArgument** = 8 - Function may modify argument values.
- **modifyArgumentAndExternal** = 12 - Function may modify argument values and external state.
- **worstDefault** = 12 - Function has all sideeffects, except for a user scenario. This is to bind functions, whith unknown sideeffects.
- **accessGlobal** = 16 - Function may access global state (variables and such).
- **invoke** = 32 - Function is using ‘invoke’, so we don’t know any additional side effects.
- **inferredSideEffects** = 56 - Mask for all sideefects, which can be inferred from the code.

10.2.3 Handled structures

`ast::ModuleLibrary`

Object which holds list of *Module* and provides access to them.

`ast::Expression`

Any expression (base class).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags

`ast::TypeDecl`

`TypeDecl.canAot() : bool()`

Returns whether the given type can be ahead-of-time compiled.

`TypeDecl.isExprType() : bool()`

Returns whether the type hierarchy contains an expression type.

`TypeDecl.isSimpleType() : bool()`

Returns whether the given type is a simple non-void type that does not require resolution at inference time.

`TypeDecl.isArray() : bool()`

Returns whether the given type is an array type.

`TypeDecl.isGoodIteratorType() : bool()`

Returns whether the given type is an iterator type.

`TypeDecl.isGoodArrayType() : bool()`

Returns whether the given type is a dynamic array type.

`TypeDecl.isGoodTableType() : bool()`

Returns whether the given type is a table type.

`TypeDecl.isGoodBlockType() : bool()`

Returns whether the given type is a block type.

`TypeDecl.isGoodFunctionType() : bool()`

Returns whether the given type is a function type.

`TypeDecl.isGoodLambdaType() : bool()`

Returns whether the given type is a lambda type.

TypeDecl.**isGoodTupleType() : bool**()

Returns whether the given type is a tuple type.

TypeDecl.**isGoodVariantType() : bool**()

Returns whether the given type is a variant type.

TypeDecl.**isVoid() : bool**()

Returns whether the given type is the void type.

TypeDecl.**isAnyType() : bool**()

Returns whether the given type is the any type, passed as vec4f via standard C++ interop.

TypeDecl.**isRef() : bool**()

Returns whether the given type is a reference value.

TypeDecl.**isRefType() : bool**()

Returns whether the given type is a reference type.

TypeDecl.**canWrite() : bool**()

Returns whether the given type can be written to.

TypeDecl.**isAotAlias() : bool**()

Returns whether the type definition contains an AOT alias type.

TypeDecl.**isShareable() : bool**()

Returns whether the given type is shareable across contexts.

TypeDecl.**isIndex() : bool**()

Returns whether the given type is an index type.

TypeDecl.**isBool() : bool**()

Returns whether the given type is a boolean type.

TypeDecl.**isInteger() : bool**()

Returns whether the given type is an integer type.

TypeDecl.**isSignedInteger() : bool**()

Returns whether the given type is a signed integer type.

TypeDecl.**isUnsignedInteger() : bool**()

Returns whether the given type is an unsigned integer type.

TypeDecl.**isSignedIntegerOrIntVec() : bool**()

Returns whether the given type is a signed integer or signed integer vector type.

TypeDecl.**isUnsignedIntegerOrIntVec() : bool**()

Returns whether the given type is an unsigned integer or unsigned integer vector type.

`TypeDecl.`**`isFloatOrDouble() : bool`**`()`

Returns whether the given type is a float or double type.

`TypeDecl.`**`isNumeric() : bool`**`()`

Returns whether the given type is a numeric type.

`TypeDecl.`**`isNumericComparable() : bool`**`()`

Returns whether the given type supports numeric comparison.

`TypeDecl.`**`isPointer() : bool`**`()`

Returns whether the given type is a pointer type.

`TypeDecl.`**`isSmartPointer() : bool`**`()`

Returns whether the given type is a smart pointer type.

`TypeDecl.`**`isVoidPointer() : bool`**`()`

Returns whether the given type is a void pointer type.

`TypeDecl.`**`isIterator() : bool`**`()`

Returns whether the given type is an iterator type.

`TypeDecl.`**`isEnum() : bool`**`()`

Returns whether the given type is an enumeration type.

`TypeDecl.`**`isEnumT() : bool`**`()`

Returns whether the base type of the given type is an enumeration type.

`TypeDecl.`**`isHandle() : bool`**`()`

Returns whether the given type is a handle type, representing a C++ type exposed to daslang via TypeAnnotation.

`TypeDecl.`**`isStructure() : bool`**`()`

Returns whether the given type is a structure type.

`TypeDecl.`**`isClass() : bool`**`()`

Returns whether the given type is a class type.

`TypeDecl.`**`isFunction() : bool`**`()`

Returns whether the given type is a function type.

`TypeDecl.`**`isTuple() : bool`**`()`

Returns whether the given type is a tuple type.

`TypeDecl.`**`isVariant() : bool`**`()`

Returns whether the given type is a variant type.

`TypeDecl.`**`sizeOf() : int`**`()`

Returns the size of the given type in bytes.

TypeDecl.**countOf() : int**()

Returns the number of elements if the given type is a fixed array, otherwise returns 1.

TypeDecl.**alignOf() : int**()

Returns the memory alignment requirement of the type in bytes.

TypeDecl.**baseSizeOf() : int**()

Returns the size of the given type in bytes, excluding fixed array dimensions.

TypeDecl.**stride() : int**()

Returns the stride size in bytes of an element in a fixed array type.

TypeDecl.**tupleSize() : int**()

Returns the size of the given tuple type in bytes.

TypeDecl.**tupleAlign() : int**()

Returns the alignment of the given tuple type in bytes.

TypeDecl.**variantSize() : int**()

Returns the size of the given variant type in bytes.

TypeDecl.**variantAlign() : int**()

Returns the alignment of the given variant type in bytes.

TypeDecl.**canCopy() : bool**()

Returns whether the given type can be copied.

TypeDecl.**canMove() : bool**()

Returns whether the given type can be moved.

TypeDecl.**canClone() : bool**()

Returns whether the given type can be cloned.

TypeDecl.**canCloneFromConst() : bool**()

Returns whether the given type can be cloned from a const instance.

TypeDecl.**canNew() : bool**()

Returns whether the given type can be heap-allocated via the new operator.

TypeDecl.**canDeletePtr() : bool**()

Returns whether the pointer to the given type can be deleted.

TypeDecl.**canDelete() : bool**()

Returns whether the given type can be deleted.

TypeDecl.**needDelete() : bool**()

Returns whether the given type requires explicit deletion.

TypeDecl.**isPod() : bool**()

Returns whether the given type is a plain old data (POD) type.

TypeDecl.**isRawPod() : bool**()

Returns whether the given type is a raw POD type containing no pointers or strings.

TypeDecl.**isNoHeapType() : bool**()

Returns whether the given type can be used without heap allocation.

TypeDecl.**isWorkhorseType() : bool**()

Returns whether the given type is a workhorse type, which is a built-in non-reference type.

TypeDecl.**isPolicyType() : bool**()

Returns whether the given type is a policy type with SimNode implementations available for it.

TypeDecl.**isVecPolicyType() : bool**()

Returns whether the given type is a vector policy type, which is any policy type other than string.

TypeDecl.**isReturnType() : bool**()

Returns whether the given type can be used as a return type, which includes anything except block.

TypeDecl.**isCtorType() : bool**()

Returns whether the given basic type is a constructor type that can be constructed via its type name, such as int(3.4).

TypeDecl.**isRange() : bool**()

Returns whether the given type is a range type.

TypeDecl.**isString() : bool**()

Returns whether the given type is a string type.

TypeDecl.**isConst() : bool**()

Returns whether the given type is const-qualified.

TypeDecl.**isFoldable() : bool**()

Returns whether the given type is foldable, such as integer or float, as opposed to pointer or array.

TypeDecl.**isAlias() : bool**()

Returns whether the type definition contains an alias type.

TypeDecl.**isAutoArrayResolved() : bool**()

Returns whether all fixed array dimensions are fully resolved with no auto or expression dimensions remaining.

TypeDecl.**isAuto() : bool**()

Returns whether the type definition contains an auto type.

TypeDecl.**isAutoOrAlias() : bool**()

Returns whether the type definition contains an auto or alias type.

TypeDecl.**isVectorType() : bool**()

Returns whether the given type is a vector type such as int2, float3, or range64.

TypeDecl.**isBitfield() : bool**()

Returns whether the given type is a bitfield type.

TypeDecl.**isLocal() : bool**()

Returns whether the given type is a local type that can be allocated on the stack.

TypeDecl.**hasClasses() : bool**()

Returns whether the type definition contains any class types.

TypeDecl.**hasNonTrivialCtor() : bool**()

Returns whether the type definition contains any non-trivial constructors.

TypeDecl.**hasNonTrivialDtor() : bool**()

Returns whether the type definition contains any non-trivial destructors.

TypeDecl.**hasNonTrivialCopy() : bool**()

Returns whether the type definition contains any non-trivial copy operations.

TypeDecl.**canBePlacedInContainer() : bool**()

Returns whether the given type can be placed in a container.

TypeDecl.**vectorBaseType() : Type**()

Returns the scalar base type of a vector type, for example float for float4.

TypeDecl.**vectorDim() : int**()

Returns the number of components in a vector type, for example 4 for float4.

TypeDecl.**canInitWithZero() : bool**()

Returns whether the given type can be initialized by zeroing its memory.

TypeDecl.**rangeBaseType() : Type**()

Returns the base type of a range type, for example int64 for range64.

TypeDecl.**unsafeInit() : bool**()

Returns whether the given type requires initialization and skipping it would be unsafe.

TypeDecl.**get_mnh() : uint64**()

Returns the mangled name hash of the given type.

Properties

- **canAot** : bool
- **isExprType** : bool
- **isSimpleType** : bool
- **isArray** : bool
- **isGoodIteratorType** : bool

- **isGoodArrayType** : bool
- **isGoodTableType** : bool
- **isGoodBlockType** : bool
- **isGoodFunctionType** : bool
- **isGoodLambdaType** : bool
- **isGoodTupleType** : bool
- **isGoodVariantType** : bool
- **isVoid** : bool
- **isAnyType** : bool
- **isRef** : bool
- **isRefType** : bool
- **canWrite** : bool
- **isAotAlias** : bool
- **isShareable** : bool
- **isIndex** : bool
- **isBool** : bool
- **isInteger** : bool
- **isSignedInteger** : bool
- **isUnsignedInteger** : bool
- **isSignedIntegerOrIntVec** : bool
- **isUnsignedIntegerOrIntVec** : bool
- **isFloatOrDouble** : bool
- **isNumeric** : bool
- **isNumericComparable** : bool
- **isPointer** : bool
- **isSmartPointer** : bool
- **isVoidPointer** : bool
- **isIterator** : bool
- **isEnum** : bool
- **isEnumT** : bool
- **isHandle** : bool
- **isStructure** : bool
- **isClass** : bool
- **isFunction** : bool
- **isTuple** : bool
- **isVariant** : bool

- **sizeOf** : int
- **countOf** : int
- **alignOf** : int
- **baseSizeOf** : int
- **stride** : int
- **tupleSize** : int
- **tupleAlign** : int
- **variantSize** : int
- **variantAlign** : int
- **canCopy** : bool
- **canMove** : bool
- **canClone** : bool
- **canCloneFromConst** : bool
- **canNew** : bool
- **canDeletePtr** : bool
- **canDelete** : bool
- **needDelete** : bool
- **isPod** : bool
- **isRawPod** : bool
- **isNoHeapType** : bool
- **isWorkhorseType** : bool
- **isPolicyType** : bool
- **isVecPolicyType** : bool
- **isReturnType** : bool
- **isCtorType** : bool
- **isRange** : bool
- **isString** : bool
- **isConst** : bool
- **isFoldable** : bool
- **isAlias** : bool
- **isAutoArrayResolved** : bool
- **isAuto** : bool
- **isAutoOrAlias** : bool
- **isVectorType** : bool
- **isBitfield** : bool
- **isLocal** : bool

- **hasClasses** : bool
- **hasNonTrivialCtor** : bool
- **hasNonTrivialDtor** : bool
- **hasNonTrivialCopy** : bool
- **canBePlacedInContainer** : bool
- **vectorBaseType** : *Type*
- **vectorDim** : int
- **canInitWithZero** : bool
- **rangeBaseType** : *Type*
- **unsafeInit** : bool
- **get_mnh** : uint64

Any type declaration.

Fields

- **baseType** : *Type* - Basic declaration type
- **structType** : *Structure*? - Structure type if baseType is Type::tStructure
- **enumType** : *Enumeration*? - Enumeration type if baseType is Type::tEnumeration
- **annotation** : *TypeAnnotation*? - Handled type if baseType is Type::tHandle
- **firstType** : smart_ptr< *TypeDecl*> - First type for compound types (like array<firstType> or table<firstType, secondType>)
- **secondType** : smart_ptr< *TypeDecl*> - Second type for compound types (like table<firstType, secondType>)
- **argTypes** : vector<smart_ptr<TypeDecl>> - Argument types for function types, tuples, variants, etc
- **argNames** : vector<das_string> - Argument names for function types
- **dim** : vector<int> - Dimensions for fixed array types
- **dimExpr** : vector<smart_ptr<Expression>> - Dimension expressions for fixed array types, when dimension is specified by expression
- **flags** : *TypeDeclFlags* - Type declaration flags
- **alias** : *das_string* - Alias name for typedefs, i.e. 'int aka MyInt' or 'MyInt'
- **at** : *LineInfo* - Location of the type declaration in the source code
- **_module** : *Module*? - Module this type belongs to

`ast::`**`Structure`**

`Structure.`**`sizeOf() : int`**`()`

Returns the size of the given type in bytes.

Properties

- **sizeOf** : int

Structure declaration.

Fields

- **name** : *das_string* - Name of the structure
- **fields** : vector<FieldDeclaration> - Field declarations of the structure
- **at** : *LineInfo* - Location of the structure declaration in the source code
- **_module** : *Module*? - Module this structure belongs to
- **parent** : *Structure*? - Parent structure, if any
- **annotations** : *AnnotationList* - List of annotations attached to this structure
- **flags** : *StructureFlags* - Structure flags

`ast::FieldDeclaration`

Structure field declaration.

Fields

- **name** : *das_string* - Name of the field
- **_type** : smart_ptr< *TypeDecl*> - Type of the field
- **init** : smart_ptr< *Expression*> - Expression for field initializer, if any
- **annotation** : *AnnotationArgumentList* - Annotations attached to this field
- **at** : *LineInfo* - Location of the field declaration in the source code
- **offset** : int - Offset of the field in the structure
- **flags** : *FieldDeclarationFlags* - Field flags

`ast::EnumEntry`

Entry in the enumeration.

Fields

- **name** : *das_string* - Name of the enumeration entry
- **cppName** : *das_string* - C++ name of the enumeration entry
- **at** : *LineInfo* - Location of the enumeration entry in the source code
- **value** : smart_ptr< *Expression*> - Value of the enumeration entry (typicall ‘ExprConst’ derivative)

`ast::Enumeration`

Enumeration declaration.

Fields

- **name** : *das_string* - Name of the enumeration
- **cppName** : *das_string* - C++ name of the enumeration
- **at** : *LineInfo* - Location of the enumeration declaration in the source code
- **list** : vector<EnumEntry> - List of entries in the enumeration
- **_module** : *Module*? - Module this enumeration belongs to
- **external** : bool - Whether this enumeration is external (defined on the C++ side)
- **baseType** : *Type* - Enumeration underlying type (int8, int16, int, or int64)

- **annotations** : *AnnotationList* - Annotations attached to this enumeration
- **isPrivate** : bool - Is this enumeration private (not visible from outside the module)

`ast::Function`

`Function.origin() : Function?()`

Returns the origin function, indicating which generic function this was instantiated from, if any.

`Function.getMangledNameHash() : uint64()`

Returns the mangled name hash of the given function.

`Function.isGeneric() : bool()`

Returns whether the given function is a generic function.

Properties

- **origin** : *Function*?
- **getMangledNameHash** : uint64
- **isGeneric** : bool

Function declaration.

Fields

- **annotations** : *AnnotationList* - Annotations attached to this function
- **name** : *das_string* - Name of the function
- **arguments** : vector<smart_ptr<Variable>> - Arguments of the function
- **result** : smart_ptr< *TypeDecl*> - Result type of the function
- **body** : smart_ptr< *Expression*> - Body expression of the function (usually 'ExprBlock' but can be optimized out on later stages)
- **index** : int - Index of the function in the 'Context'
- **totalStackSize** : uint - Stack size required for this function
- **totalGenLabel** : int - Number of generated labels in the jump table (for the generator)
- **at** : *LineInfo* - Location of the function in the source code
- **atDecl** : *LineInfo* - Location of the function declaration in the source code
- **_module** : *Module*? - Module this function belongs to
- **classParent** : *Structure*? - Parent structure if this is a method
- **flags** : *FunctionFlags* - Function flags
- **moreFlags** : *MoreFunctionFlags* - More function flags
- **sideEffectFlags** : *FunctionSideEffectFlags* - Function side effect flags
- **inferStack** : vector<InferHistory> - Inference history
- **fromGeneric** : smart_ptr< *Function*> - If this function was instantiated from a generic function, pointer to the generic function
- **hash** : uint64 - Hash of the function signature
- **aotHash** : uint64 - Hash of the function signature for AOT purposes

ast::**BuiltInFunction**

Bindings for the 'BuiltInFunction', which is used for the builtin (bound) functions in Daslang.

Fields

- **annotations** : *AnnotationList* - Annotations attached to this function
- **name** : *das_string* - Name of the function
- **arguments** : vector<smart_ptr<Variable>> - Arguments of the function
- **result** : smart_ptr< *TypeDecl*> - Result type of the function
- **body** : smart_ptr< *Expression*> - Body expression of the function (null just about every time for the builtins)
- **index** : int - Index of the function in the 'Context'
- **totalStackSize** : uint - Stack size required for this function
- **totalGenLabel** : int - Number of generated labels in the jump table (for the generator)
- **at** : *LineInfo* - Location of the function in the source code
- **atDecl** : *LineInfo* - Location of the function declaration in the source code
- **_module** : *Module*? - Module this function belongs to
- **classParent** : *Structure*? - Parent structure if this is a method
- **flags** : *FunctionFlags* - Function flags
- **moreFlags** : *MoreFunctionFlags* - More function flags
- **sideEffectFlags** : *FunctionSideEffectFlags* - Function side effect flags
- **inferStack** : vector<InferHistory> - Inference history
- **fromGeneric** : smart_ptr< *Function*> - If this function was instantiated from a generic function, pointer to the generic function
- **hash** : uint64 - Hash of the function signature
- **aotHash** : uint64 - Hash of the function signature for AOT purposes
- **cppName** : *das_string* - C++ function name.

ast::**ExternalFnBase**

Base class for external function bindings. Bindings for the 'BuiltInFunction', which is used for the builtin (bound) functions in Daslang.

Fields

- **annotations** : *AnnotationList* - Annotations attached to this function
- **name** : *das_string* - Name of the function
- **arguments** : vector<smart_ptr<Variable>> - Arguments of the function
- **result** : smart_ptr< *TypeDecl*> - Result type of the function
- **body** : smart_ptr< *Expression*> - Body expression of the function (null just about every time for the external functions)
- **index** : int - Index of the function in the 'Context'
- **totalStackSize** : uint - Stack size required for this function

- **totalGenLabel** : int - Number of generated labels in the jump table (for the generator)
- **at** : *LineInfo* - Location of the function in the source code
- **atDecl** : *LineInfo* - Location of the function declaration in the source code
- **_module** : *Module*? - Module this function belongs to
- **classParent** : *Structure*? - Parent structure if this is a method
- **flags** : *FunctionFlags* - Function flags
- **moreFlags** : *MoreFunctionFlags* - More function flags
- **sideEffectFlags** : *FunctionSideEffectFlags* - Function side effect flags
- **inferStack** : vector<InferHistory> - Inference history
- **fromGeneric** : smart_ptr< *Function*> - If this function was instantiated from a generic function, pointer to the generic function
- **hash** : uint64 - Hash of the function signature
- **aotHash** : uint64 - Hash of the function signature for AOT purposes
- **cppName** : *das_string* - C++ function name.

`ast::`**`InferHistory`**

Generic function infer history.

Fields

- **at** : *LineInfo* - Location of the function in the source code
- **func** : *Function*? - Function being inferred

`ast::`**`Variable`**

`Variable.`**`isAccessUnused() : bool`**`()`

Returns whether the given variable is never accessed in the code.

`Variable.`**`getMangledNameHash() : uint64`**`()`

Returns the mangled name hash of the given function.

Properties

- **isAccessUnused** : bool
- **getMangledNameHash** : uint64

Variable declaration.

Fields

- **name** : *das_string* - Name of the variable
- **_aka** : *das_string* - Alternative name of the variable
- **_type** : smart_ptr< *TypeDecl*> - Type of the variable
- **init** : smart_ptr< *Expression*> - Initializer expression for the variable, if any
- **source** : smart_ptr< *Expression*> - If its an iterator variable for the for loop, source expression being iterated over
- **at** : *LineInfo* - Location of the variable declaration in the source code

- **index** : int - Index of the variable in the global variable list (for global variables)
- **stackTop** : uint - Stack top offset for local variables
- **_module** : *Module*? - Module this variable belongs to
- **initStackSize** : uint - Stack size required to evaluate the initializer expression (for global variables)
- **flags** : *VariableFlags* - Variable flags
- **access_flags** : *VariableAccessFlags* - Variable access flags
- **annotation** : *AnnotationArgumentList* - Annotations attached to this variable

`ast::`**`AstContext`**

Lexical context for the particular expression.

Fields

- **func** : smart_ptr< *Function*> - Function this expression belongs to
- **_loop** : vector<smart_ptr<Expression>> - Loop stack
- **blocks** : vector<smart_ptr<Expression>> - Stack of active blocks
- **scopes** : vector<smart_ptr<Expression>> - Stack of active scopes
- **_with** : vector<smart_ptr<Expression>> - Stack of active 'with' expressions

`ast::`**`ExprBlock`**

Any block expression, including regular blocks and all types of closures.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **list** : vector<smart_ptr<Expression>> - List of expressions in the main body of the block
- **finalList** : vector<smart_ptr<Expression>> - List of expressions in the 'finally' section of the block
- **returnType** : smart_ptr< *TypeDecl*> - Declared return type of the block, if any (for closures)
- **arguments** : vector<smart_ptr<Variable>> - List of arguments for the block (for closures)
- **stackTop** : uint - Stack top offset for the block declaration
- **stackVarTop** : uint - Where variables of the block start on the stack
- **stackVarBottom** : uint - Where variables of the block end on the stack
- **stackCleanVars** : vector<pair`uint`uint> - Variables which are to be zeroed, if there is 'finally' section of the block. If there is 'inscope' variable after the return, it should be zeroed before entering the block.

- **maxLabelIndex** : int - Maximum label index used in this block (for goto statements)
- **annotations** : *AnnotationList* - AnnotationList - Annotations attached to this block
- **annotationData** : uint64 - Opaque data associated with block
- **annotationDataSid** : uint64 - Opaque data source unique-ish id associated with block
- **blockFlags** : *ExprBlockFlags* - Block expression flags
- **inFunction** : *Function*? - Which function this block belongs to

`ast::ExprLet`

Local variable declaration (*let v = expr;*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **variables** : vector<smart_ptr<Variable>> - List of variables being declared in this *let* expression
- **atInit** : *LineInfo* - Location of the initializer expression in source code
- **letFlags** : *ExprLetFlags* - Properties of the *ExprLet* object.

`ast::ExprStringBuilder`

String builder expression ("blah{blah1}blah2").

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **elements** : vector<smart_ptr<Expression>> - List of expressions that make up the string builder (literals and expressions)
- **stringBuilderFlags** : *StringBuilderFlags* - Flags specific to string builder expressions

`ast::MakeFieldDecl`

Part of *ExprMakeStruct*, declares single field (*a = expr* or *a <- expr* etc)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **name** : *das_string* - Name of the field being assigned

- **value** : smart_ptr< *Expression*> - Initializer expression for the field
- **tag** : smart_ptr< *Expression*> - Tag associated with the field, if any
- **flags** : *MakeFieldDeclFlags* - Flags specific to this field declaration

`ast::`**`ExprNamedCall`**

Named call (*call([argname1=expr1, argname2=expr2])*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the named call
- **nonNamedArguments** : vector<smart_ptr<Expression>> - Non-named arguments passed to the call
- **arguments** : *MakeStruct* - Named arguments passed to the call
- **argumentsFailedToInfer** : bool - Whether any arguments failed to infer their types

`ast::`**`ExprLooksLikeCall`**

Anything which looks like call (*call(expr1,expr2)*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call
- **arguments** : vector<smart_ptr<Expression>> - List of arguments passed to the call
- **argumentsFailedToInfer** : bool - Whether any arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code

`ast::`**`ExprCallFunc`**

Actual function call (*func(expr1,…)*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the called function
- **arguments** : vector<smart_ptr<Expression>> - Arguments passed to the function
- **argumentsFailedToInfer** : bool - Whether any arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Pointer to the function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed

ast::**ExprNew**

New expression (*new Foo*, *new Bar(expr1..)*, but **NOT** *new [[Foo …]]*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the new expression
- **arguments** : vector<smart_ptr<Expression>> - List of arguments passed to the constructor
- **argumentsFailedToInfer** : bool - Whether any arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Pointer to the constructor function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **typeexpr** : smart_ptr< *TypeDecl*> - Type expression for the type being constructed
- **initializer** : bool - Whether there is an initializer for the new expression, or it’s just default construction

ast::**ExprCall**

Anything which looks like call (*call(expr1,expr2)*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)

- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call
- **arguments** : vector<smart_ptr<Expression>> - List of arguments passed to the function
- **argumentsFailedToInfer** : bool - Whether any arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Pointer to the function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **doesNotNeedSp** : bool - If the call does not need stack pointer
- **cmresAlias** : bool - If the call uses CMRES (Copy or Move result) aliasing, i.e would need temporary
- **notDiscarded** : bool - If the call result is not discarded

ast::**ExprPtr2Ref**

Pointer dereference (**expr* or *deref(expr)*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Expression being dereferenced
- **unsafeDeref** : bool - If true, skip runtime null-pointer check
- **assumeNoAlias** : bool - If true, assume no aliasing occurs

ast::**ExprNullCoalescing**

Null coalescing (*expr1 ?? default_value*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Expression being coalesced

- **unsafeDeref** : bool - If true, skip runtime null-pointer check
- **assumeNoAlias** : bool - Assume no aliasing occurs
- **defaultValue** : smart_ptr< *Expression*> - Default value expression

`ast::`**`ExprAt`**

Index lookup (*expr[expr1]*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Subexpression being indexed
- **index** : smart_ptr< *Expression*> - Index expression
- **atFlags** : *ExprAtFlags* - Flags specific to *ExprAt* expressions

`ast::`**`ExprSafeAt`**

Safe index lookup (*expr?[expr1]*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Subexpression being indexed
- **index** : smart_ptr< *Expression*> - Index expression
- **atFlags** : *ExprAtFlags* - Flags specific to *ExprAt* expressions

`ast::`**`ExprIs`**

Is expression for variants and such (*expr is Foo*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags

- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Subexpression being checked
- **typeexpr** : smart_ptr< *TypeDecl*> - Type being checked against

`ast::`**`ExprOp`**

Compilation time only base class for any operator.

`ast::`**`ExprOp2`**

Two operand operator (*expr1 + expr2*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments (unused)
- **argumentsFailedToInfer** : bool - If arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **left** : smart_ptr< *Expression*> - Left operand expression
- **right** : smart_ptr< *Expression*> - Right operand expression

`ast::`**`ExprOp3`**

Three operand operator (*cond ? expr1 : expr2*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments (unused)

- **argumentsFailedToInfer** : bool - If arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **subexpr** : smart_ptr< *Expression*> - Condition expression
- **left** : smart_ptr< *Expression*> - Left operand expression
- **right** : smart_ptr< *Expression*> - Right operand expression

ast::**ExprCopy**

Copy operator (*expr1 = expr2*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments (unused)
- **argumentsFailedToInfer** : bool - If arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **left** : smart_ptr< *Expression*> - Left operand expression
- **right** : smart_ptr< *Expression*> - Right operand expression
- **copy_flags** : *CopyFlags* - Flags specific to copy operation

ast::**ExprMove**

Move operator (*expr1 <- expr2*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags

- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments (unused)
- **argumentsFailedToInfer** : bool - If arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **left** : smart_ptr< *Expression*> - Left operand expression
- **right** : smart_ptr< *Expression*> - Right operand expression
- **move_flags** : *MoveFlags* - Flags specific to move operation

`ast::`**`ExprClone`**

Clone operator (*expr1 := expr2*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the call (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments (unused)
- **argumentsFailedToInfer** : bool - If arguments failed to infer their types
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function being called, if resolved
- **stackTop** : uint - Stack top at the point of call, if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **left** : smart_ptr< *Expression*> - Left operand expression
- **right** : smart_ptr< *Expression*> - Right operand expression

`ast::`**`ExprWith`**

With section (*with expr {your; block; here}*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **_with** : smart_ptr< *Expression*> - The expression to be used as the context for the with block
- **body** : smart_ptr< *Expression*> - The body of the with block

ast::**ExprAssume**

Assume expression (*assume name = expr*) or (*typedef name = type*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **alias** : *das_string* - The alias name for the assume expression
- **subexpr** : smart_ptr< *Expression*> - The expression being aliased, if specified
- **assumeType** : smart_ptr< *TypeDecl*> - The type being assumed, if specified

ast::**ExprWhile**

While loop (*while expr {your; block; here;}*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **cond** : smart_ptr< *Expression*> - The condition expression
- **body** : smart_ptr< *Expression*> - The body of the while loop

ast::**ExprTryCatch**

Try-recover expression (*try {your; block; here;} recover {your; recover; here;}*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **try_block** : smart_ptr< *Expression*> - The try block
- **catch_block** : smart_ptr< *Expression*> - The recover block

ast::**ExprIfThenElse**

If-then-else expression (*if expr1 {your; block; here;} else {your; block; here;}*) including *static_if* 's.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **cond** : smart_ptr< *Expression*> - The condition expression
- **if_true** : smart_ptr< *Expression*> - The 'then' block expression
- **if_false** : smart_ptr< *Expression*> - The 'else' block expression
- **if_flags** : *IfFlags* - Flags specific to if-then-else expressions

ast::**ExprFor**

For loop (*for expr1 in expr2 {your; block; here;}*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **iterators** : vector<das_string> - Names of the iterator variables
- **iteratorsAka** : vector<das_string> - Aliases for the iterator variables
- **iteratorsAt** : vector<LineInfo> - Line information for each iterator
- **iteratorsTags** : vector<smart_ptr<Expression>> - Tags associated with each iterator
- **iteratorsTupleExpansion** : vector<uint8> - Tuple expansion flags for iterators
- **iteratorVariables** : vector<smart_ptr<Variable>> - Variables associated with each iterator

- **sources** : vector<smart_ptr<Expression>> - Source expressions to iterate over
- **body** : smart_ptr< *Expression*> - The body of the for loop
- **visibility** : *LineInfo* - Line information for visibility of the iterators
- **allowIteratorOptimization** : bool - Whether iterator optimization is allowed
- **canShadow** : bool - Whether shadowing is allowed, i.e. if the iterator names can shadow outer scope variables

ast::**ExprMakeLocal**

Any make expression (*ExprMakeBlock*, *ExprMakeTuple*, *ExprMakeVariant*, *ExprMakeStruct*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **makeType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Stack top offset for the data, if applicable
- **extraOffset** : uint - Extra offset for the data, if applicable. If part of the larger initialization, extra offset is that
- **makeFlags** : *ExprMakeLocalFlags* - Flags specific to make-local expressions

ast::**ExprMakeStruct**

Make structure expression (*[[YourStruct v1=expr1elem1, v2=expr2elem1, …; v1=expr1elem2, …]]*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **makeType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Stack top offset for the data, if applicable
- **extraOffset** : uint - Extra offset for the data, if applicable. If part of the larger initialization, extra offset is that
- **makeFlags** : *ExprMakeLocalFlags* - Flags specific to make-local expressions
- **structs** : vector<smart_ptr<MakeStruct>> - Array of structures being made

- **_block** : smart_ptr< *Expression*> - Optional block expression to run after the struct is made
- **constructor** : *Function*? - Constructor function to call, if any
- **makeStructFlags** : *ExprMakeStructFlags* - Flags specific to make-struct expressions

`ast::`**`ExprMakeVariant`**

Make variant expression (*[YourVariant variantName=expr1]*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **makeType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Stack top offset for the data, if applicable
- **extraOffset** : uint - Extra offset for the data, if applicable. If part of the larger initialization, extra offset is that
- **makeFlags** : *ExprMakeLocalFlags* - Flags specific to make-local expressions
- **variants** : vector<smart_ptr<MakeFieldDecl>> - Array of variants being made

`ast::`**`ExprMakeArray`**

Make array expression (*[[auto 1;2;3]]* or *[{auto "foo";"bar"}]* for static and dynamic arrays accordingly).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **makeType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Stack top offset for the data, if applicable
- **extraOffset** : uint - Extra offset for the data, if applicable. If part of the larger initialization, extra offset is that
- **makeFlags** : *ExprMakeLocalFlags* - Flags specific to make-local expressions
- **recordType** : smart_ptr< *TypeDecl*> - Type of the array elements
- **values** : vector<smart_ptr<Expression>> - Array of expressions for the elements
- **gen2** : bool - If gen2 syntax is used (i.e. *[. . .]* instead of *[[. . .]]*)

`ast::`**`ExprMakeTuple`**

Make tuple expression (*[[auto f1,f2,f3]]*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **makeType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Stack top offset for the data, if applicable
- **extraOffset** : uint - Extra offset for the data, if applicable. If part of the larger initialization, extra offset is that
- **makeFlags** : *ExprMakeLocalFlags* - Flags specific to make-local expressions
- **recordType** : smart_ptr< *TypeDecl*> - Type of the array elements
- **values** : vector<smart_ptr<Expression>> - Array of expressions for the elements
- **gen2** : bool - If gen2 syntax is used (i.e. *[. . .]* instead of *[[. . .]]*)
- **isKeyValue** : bool - If key-value syntax is used (i.e. *[key=>val; key2=>val2]*)

`ast::`**`ExprArrayComprehension`**

Array comprehension (*[for (x in 0..3); x]*, *[iterator for (y in range(100)); x*2; where (x!=13)]]* for arrays or generators accordingly).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **exprFor** : smart_ptr< *Expression*> - The 'for' expression
- **exprWhere** : smart_ptr< *Expression*> - The 'where' expression
- **subexpr** : smart_ptr< *Expression*> - The subexpression
- **generatorSyntax** : bool - If generator syntax is used (i.e. *[iterator for . . .]* instead of *[for]*)
- **tableSyntax** : bool - If table syntax is used (i.e. *{for . . . }* instead of *[for]*)

`ast::TypeInfoMacro`

Compilation time only structure which holds live information about typeinfo expression for the specific macro.

Fields

- **name** : *das_string* - The name of the macro
- **_module** : *Module*? - The module where the macro is defined

`ast::ExprTypeInfo`

typeinfo() expression (*typeinfo dim(a)*, *typeinfois_ref_type<int&>()*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **trait** : *das_string* - The trait name
- **subexpr** : smart_ptr< *Expression*> - The expression being queried for type information
- **typeexpr** : smart_ptr< *TypeDecl*> - The type expression being queried for type information
- **subtrait** : *das_string* - The sub-trait name
- **extratrait** : *das_string* - The extra trait name
- **macro** : *TypeInfoMacro*? - The macro associated with the typeinfo expression

`ast::ExprTypeDecl`

typedecl() expression (*typedecl(1+2)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **typeexpr** : smart_ptr< *TypeDecl*> - The type expression being queried for type information

`ast::ExprLabel`

Label (*label 13:*)

Fields

- **at** : *LineInfo* - Location of the expression in source code

- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **labelName** : int - The label name
- **comment** : *das_string* - The label comment

`ast::`**`ExprGoto`**

Goto expression (*goto label 13*, *goto x*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **labelName** : int - Label to go to, if specified
- **subexpr** : smart_ptr< *Expression*> - Expression evaluating to label to go to, if specified

`ast::`**`ExprRef2Value`**

Compilation time only structure which holds reference to value conversion for the value types, i.e. goes from int& to int and such.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The sub-expression being converted from reference to value

`ast::`**`ExprRef2Ptr`**

Addr expresion (*addr(expr)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The sub-expression being converted from pointer to reference

`ast::`**`ExprAddr`**

Function address (*@@foobarfunc* or *@@foobarfunc<(int;int):bool>*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **target** : *das_string* - Name of the function being referenced
- **funcType** : smart_ptr< *TypeDecl*> - Type of the function being referenced
- **func** : *Function*? - Function being referenced (if resolved)

`ast::`**`ExprAssert`**

Assert expression (*assert(x<13)*, or *assert(x<13, "x is too big")*, or *verify(foo()!=0)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the asserted expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the assert expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the assert is used
- **isVerify** : bool - Whether the assert is a verify expression (verify expressions have to have sideeffects, assert expressions cant)

ast::**ExprQuote**

Compilation time expression which holds its subexpressions but does not infer them (*quote() <| x+5*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the query expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the query expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the query is used

ast::**ExprStaticAssert**

Static assert expression (*static_assert(x<13)* or *static_assert(x<13, "x is too big")*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the static_assert expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the static_assert expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the static_assert is used

ast::**ExprDebug**

Debug expression (*debug(x)* or *debug(x,"x=")*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags

- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the debug expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the debug expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the debug is used

`ast::ExprInvoke`

`ExprInvoke.isCopyOrMove() : bool()`

Returns whether the given invoke expression requires a copy or move of a reference type.

Properties

- **isCopyOrMove** : bool

Invoke expression (*invoke(fn)* or *invoke(lamb, arg1, arg2, …)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the invoke expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the invoke expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the invoke is used
- **stackTop** : uint - Stack top for invoke, if applicable
- **doesNotNeedSp** : bool - Does not need stack pointer
- **isInvokeMethod** : bool - Is invoke of class method
- **cmresAlias** : bool - If true, then CMRES aliasing is allowed for this invoke (and stack will be allocated)

`ast::ExprErase`

Erase expression (*erase(tab,key)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags

- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the erase expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the erase expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the erase is used

`ast::`**`ExprSetInsert`**

Set insert expression, i.e. `tab |> insert(key)`.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the set-insert expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the set-insert expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the set-insert is used

`ast::`**`ExprFind`**

Find expression (*find(tab,key)* <| *{ your; block; here; }*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the find expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the find expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the find is used

`ast::`**`ExprKeyExists`**

Key exists expression (*key_exists(tab,key)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the key-exists expression
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the key-exists expression
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the key-exists is used

ast::**ExprAscend**

New expression for ExprMakeLocal (*new [[Foo fld=val,…]]* or *new [[Foo() fld=…]]*, but **NOT** *new Foo()*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Subexpression being ascended (newed)
- **ascType** : smart_ptr< *TypeDecl*> - Type being made
- **stackTop** : uint - Location on the stack where the temp object is created, if necessary
- **ascendFlags** : *ExprAscendFlags* - Flags specific to *ExprAscend* expressions

ast::**ExprCast**

Any cast expression (*cast<int> a*, *upcast<Foo> b* or *reinterpret<Bar?> c*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - Subexpression being cast
- **castType** : smart_ptr< *TypeDecl*> - Type to which the expression is being cast

- **castFlags** : *ExprCastFlags* - Flags specific to *ExprCast* expressions

ast::**ExprDelete**

Delete expression (*delete blah*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The expression being deleted
- **sizeexpr** : smart_ptr< *Expression*> - The size expression for deleting classes. This one determines how big instance is to be deleted.
- **native** : bool - True if the delete is native, and not to be expanded at compilation time.

ast::**ExprVar**

Variable access (*foo*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - The name of the variable
- **variable** : smart_ptr< *Variable*> - The variable being accessed
- **pBlock** : *ExprBlock*? - The block in which the variable is accessed (if any)
- **argumentIndex** : int - The argument index of the variable (if variable is an argument of a function or a block)
- **varFlags** : *ExprVarFlags* - The flags of the variable

ast::**ExprTag**

Compilation time only tag expression, used for reification. For example $c(….).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The subexpression of the tag
- **value** : smart_ptr< *Expression*> - Value of the tag
- **name** : *das_string* - Name of the tag

`ast::`**`ExprSwizzle`**

Vector swizzle operation (*vec.xxy* or *vec.y*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Value being swizzled
- **mask** : *das_string* - Swizzle mask
- **fields** : vector<uint8> - Swizzle fields
- **fieldFlags** : *ExprSwizzleFieldFlags* - Flags specific to *ExprSwizzle* expressions

`ast::`**`ExprField`**

`ExprField.`**`field() : FieldDeclaration?`**`()`

Returns a pointer to the named field of a structure, or null if the field does not exist or the type is not a structure.

Properties

- **field** : *FieldDeclaration*?

Field lookup (*foo.bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Subexpression whose field is being accessed

- **name** : *das_string* - Name of the field being accessed
- **atField** : *LineInfo* - Location of the field access in source code
- **fieldIndex** : int - Index of the field in the type's field list
- **annotation** : smart_ptr< *TypeAnnotation*> - Type annotation for the field
- **derefFlags** : *ExprFieldDerefFlags* - Flags for dereferencing operations
- **fieldFlags** : *ExprFieldFieldFlags* - Flags specific to field access expressions

`ast::`**`ExprSafeField`**

Safe field lookup (*foo?.bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Subexpression whose field is being accessed
- **name** : *das_string* - Name of the field being accessed
- **atField** : *LineInfo* - Location of the field access in source code
- **fieldIndex** : int - Index of the field in the type's field list
- **annotation** : smart_ptr< *TypeAnnotation*> - Type annotation for the field
- **derefFlags** : *ExprFieldDerefFlags* - Flags for dereferencing operations
- **fieldFlags** : *ExprFieldFieldFlags* - Flags specific to field access expressions
- **skipQQ** : bool - If true the subexpression is already a pointer and no additional dereference is needed

`ast::`**`ExprIsVariant`**

Is expression (*foo is bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Subexpression whose field is being accessed
- **name** : *das_string* - Name of the field being accessed

- **atField** : *LineInfo* - Location of the field access in source code
- **fieldIndex** : int - Index of the field in the type's field list
- **annotation** : smart_ptr< *TypeAnnotation*> - Type annotation for the field
- **derefFlags** : *ExprFieldDerefFlags* - Flags for dereferencing operations
- **fieldFlags** : *ExprFieldFieldFlags* - Flags specific to field access expressions

`ast::`**`ExprAsVariant`**

As expression (*foo as bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Subexpression whose field is being accessed
- **name** : *das_string* - Name of the field being accessed
- **atField** : *LineInfo* - Location of the field access in source code
- **fieldIndex** : int - Index of the field in the type's field list
- **annotation** : smart_ptr< *TypeAnnotation*> - Type annotation for the field
- **derefFlags** : *ExprFieldDerefFlags* - Flags for dereferencing operations
- **fieldFlags** : *ExprFieldFieldFlags* - Flags specific to field access expressions

`ast::`**`ExprSafeAsVariant`**

Safe as expression (*foo? as bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **value** : smart_ptr< *Expression*> - Subexpression whose field is being accessed
- **name** : *das_string* - Name of the field being accessed
- **atField** : *LineInfo* - Location of the field access in source code
- **fieldIndex** : int - Index of the field in the type's field list

- **annotation** : smart_ptr< *TypeAnnotation*> - Type annotation for the field
- **derefFlags** : *ExprFieldDerefFlags* - Flags for dereferencing operations
- **fieldFlags** : *ExprFieldFieldFlags* - Flags specific to field access expressions
- **skipQQ** : bool - If true the subexpression is already a pointer and no additional dereference is needed

`ast::ExprOp1`

Single operator expression (*+a* or *-a* or *!a* or *~a*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the operator (unused)
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the operator (unused)
- **argumentsFailedToInfer** : bool - Whether arguments failed to infer
- **atEnclosure** : *LineInfo* - Location of the expression in source code
- **func** : *Function*? - Function associated with the expression
- **stackTop** : uint - Stack top position if temporary variable allocation is needed
- **op** : *das_string* - Name of the operator
- **subexpr** : smart_ptr< *Expression*> - That one argument of the operator

`ast::ExprReturn`

Return expression (*return* or *return foo*, or *return <- foo*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The expression being returned (if any)
- **returnFlags** : *ExprReturnFlags* - Return flags
- **stackTop** : uint - Stack top position if temporary variable allocation is needed
- **refStackTop** : uint - Reference stack top position if temporary variable allocation is needed

- **returnFunc** : *Function*? - Function associated with the return expression
- **_block** : *ExprBlock*? - Block associated with the return expression

`ast::ExprYield`

Yield expression (*yield foo* or *yield <- bar*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **subexpr** : smart_ptr< *Expression*> - The expression being yielded (never empty)
- **returnFlags** : *ExprYieldFlags* - Yield flags

`ast::ExprBreak`

Break expression (*break*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags

`ast::ExprContinue`

Continue expression (*continue*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags

ast::**ExprConst**

Compilation time constant expression base class

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression

ast::**ExprFakeContext**

Compilation time only fake context expression. Will simulate as current evaluation *Context*.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::fakeContext)

ast::**ExprFakeLineInfo**

ExprFakeLineInfo.**getValue() : void?**()

Returns the constant value stored in this expression node.

Properties

- **getValue** : void?

Compilation time only fake lineinfo expression. Will simulate as current file and line *LineInfo*.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags

- **baseType** : *Type* - Base type of the constant expression (Type::fakeLineInfo)
- **value** : void? - Pointer to the LineInfo, as void?

`ast::`**`ExprConstPtr`**

`ExprConstPtr.`**`getValue() : void?`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : void?

Null (*null*). Technically can be any other pointer, but it is used for nullptr.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression
- **value** : void? - Pointer value. Typically this is 'null' constant, so the value is zero.

`ast::`**`ExprConstInt8`**

`ExprConstInt8.`**`getValue() : int8`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int8

Holds int8 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt8)
- **value** : int8 - Value of the constant expression

`ast::`**`ExprConstInt16`**

`ExprConstInt16.`**`getValue() : int16`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int16

Holds int16 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt16)
- **value** : int16 - Value of the constant expression

`ast::`**`ExprConstInt64`**

`ExprConstInt64.`**`getValue() : int64`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int64

Holds int64 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt64)
- **value** : int64 - Value of the constant expression

`ast::`**`ExprConstInt`**

`ExprConstInt.`**`getValue() : int`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int

Holds int constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt)
- **value** : int - Value of the constant expression

`ast::`**`ExprConstInt2`**

`ExprConstInt2.`**`getValue() : int2`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int2

Holds int2 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt2)
- **value** : int2 - Value of the constant expression

`ast::`**`ExprConstInt3`**

`ExprConstInt3.`**`getValue() : int3`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int3

Holds int3 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt3)
- **value** : int3 - Value of the constant expression

`ast::`**`ExprConstInt4`**

`ExprConstInt4.`**`getValue() : int4`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : int4

Holds int4 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tInt4)
- **value** : int4 - Value of the constant expression

`ast::`**`ExprConstUInt8`**

`ExprConstUInt8.`**`getValue() : uint8`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint8

Holds uint8 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags

- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt8)
- **value** : uint8 - Value of the constant expression

`ast::`**`ExprConstUInt16`**

`ExprConstUInt16.`**`getValue() : uint16`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint16

Holds uint16 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt16)
- **value** : uint16 - Value of the constant expression

`ast::`**`ExprConstUInt64`**

`ExprConstUInt64.`**`getValue() : uint64`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint64

Holds uint64 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt64)
- **value** : uint64 - Value of the constant expression

`ast::`**`ExprConstUInt`**

`ExprConstUInt.`**`getValue() : uint`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint

Holds uint constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt)
- **value** : uint - Value of the constant expression

`ast::`**`ExprConstUInt2`**

`ExprConstUInt2.`**`getValue() : uint2`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint2

Holds uint2 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt2)
- **value** : uint2 - Value of the constant expression

`ast::`**`ExprConstUInt3`**

ExprConstUInt3.**getValue() : uint3**()

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint3

Holds uint3 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt3)
- **value** : uint3 - Value of the constant expression

ast::**ExprConstUInt4**

ExprConstUInt4.**getValue() : uint4**()

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint4

Holds uint4 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tUInt4)
- **value** : uint4 - Value of the constant expression

ast::**ExprConstRange**

ExprConstRange.**getValue() : range**()

Returns the constant value stored in this expression node.

Properties

- **getValue** : range

Holds range constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tRange)
- **value** : range - Value of the constant expression

`ast::`**`ExprConstURange`**

`ExprConstURange.`**`getValue() : urange`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : urange

Holds urange constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tURange)
- **value** : urange - Value of the constant expression

`ast::`**`ExprConstRange64`**

`ExprConstRange64.`**`getValue() : range64`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : range64

Holds range64 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tRange64)
- **value** : range64 - Value of the constant expression

`ast::`**`ExprConstURange64`**

`ExprConstURange64.`**`getValue() : urange64`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : urange64

Holds urange64 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tURange64)
- **value** : urange64 - Value of the constant expression

`ast::`**`ExprConstFloat`**

`ExprConstFloat.`**`getValue() : float`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : float

Holds float constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags

- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tFloat)
- **value** : float - Value of the constant expression

`ast::`**`ExprConstFloat2`**

`ExprConstFloat2.`**`getValue() : float2`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : float2

Holds float2 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tFloat2)
- **value** : float2 - Value of the constant expression

`ast::`**`ExprConstFloat3`**

`ExprConstFloat3.`**`getValue() : float3`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : float3

Holds float3 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tFloat3)
- **value** : float3 - Value of the constant expression

`ast::`**`ExprConstFloat4`**

`ExprConstFloat4.`**`getValue() : float4`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : float4

Holds float4 constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tFloat4)
- **value** : float4 - Value of the constant expression

`ast::`**`ExprConstDouble`**

`ExprConstDouble.`**`getValue() : double`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : double

Holds double constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tDouble)
- **value** : double - Value of the constant expression

`ast::`**`ExprConstBool`**

`ExprConstBool.getValue() : bool()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : bool

Holds bool constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tBool)
- **value** : bool - Value of the constant expression

`ast::CaptureEntry`

Single entry in lambda capture.

Fields

- **name** : *das_string* - Name of the captured variable
- **mode** : *CaptureMode* - How the variable is captured (by value, by reference, etc.)

`ast::ExprMakeBlock`

Any closure. Holds block as well as capture information in *CaptureEntry*.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **_capture** : vector<CaptureEntry> - List of captured variables
- **_block** : smart_ptr< *Expression*> - The block expression
- **stackTop** : uint - Stack top for the block
- **mmFlags** : *ExprMakeBlockFlags* - Expression generation flags
- **aotFunctorName** : *das_string* - Name of the AOT functor (if applicable)

ast::**ExprMakeGenerator**

Generator closure (*generator<int>* or *generator<Foo&>*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the AOT functor (if applicable)
- **arguments** : vector<smart_ptr<Expression>> - Arguments passed to the generator
- **argumentsFailedToInfer** : bool - Whether arguments failed to infer
- **atEnclosure** : *LineInfo* - Location of the enclosure
- **iterType** : smart_ptr< *TypeDecl*> - Iterator type, i.e. type of values produced by the generated iterator
- **_capture** : vector<CaptureEntry> - List of captured variables

ast::**ExprMemZero**

Memzero (*memzero(expr)*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **name** : *das_string* - Name of the memzero call
- **arguments** : vector<smart_ptr<Expression>> - Arguments of the memzero call
- **argumentsFailedToInfer** : bool - Whether the arguments failed to infer types
- **atEnclosure** : *LineInfo* - Location of the enclosure where the memzero is used

ast::**ExprConstEnumeration**

Holds enumeration constant, both type and entry (*Foo bar*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression

- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tEnumeration, Type::tEnumeration8, Type::tEnumeration16, or Type::tEnumeration64)
- **enumType** : smart_ptr< *Enumeration*> - Enumeration type declaration
- **value** : *das_string* - Value of the constant expression

`ast::`**`ExprConstBitfield`**

`ExprConstBitfield.`**`getValue() : uint64`**`()`

Returns the constant value stored in this expression node.

Properties

- **getValue** : uint64

Holds bitfield constant (*Foo bar*).

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **baseType** : *Type* - Base type of the constant expression (Type::tBitfield, Type::tBitfield8, Type::tBitfield16, or Type::tBitfield64)
- **value** : bitfield<> - Value of the constant expression
- **bitfieldType** : smart_ptr< *TypeDecl*> - Type declaration of the bitfield

`ast::`**`ExprConstString`**

Holds string constant.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e “ExprConstant”, “ExprCall”, etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags

- **baseType** : *Type* - Base type of the constant expression (Type::tString)
- **value** : *das_string* - Value of the constant expression

`ast::ExprUnsafe`

Unsafe expression (*unsafe(addr(x))*)

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **body** : smart_ptr< *Expression*> - Body expression that is marked as unsafe

`ast::VisitorAdapter`

Adapter for the *AstVisitor* interface.

`ast::FunctionAnnotation`

Adapter for the *AstFunctionAnnotation*.

`ast::StructureAnnotation`

Adapter for the *AstStructureAnnotation*.

`ast::EnumerationAnnotation`

Adapter for the *AstEnumerationAnnotation*.

`ast::PassMacro`

Adapter for the *AstPassMacro*.

Fields

- **name** : *das_string* - Name of the macro

`ast::ReaderMacro`

Adapter for the *AstReaderMacro*.

Fields

- **name** : *das_string* - Name of the macro
- **_module** : *Module*? - Module where the macro is defined

`ast::CommentReader`

Adapter for the *AstCommentReader*.

`ast::CallMacro`

Adapter for the *AstCallMacro*.

Fields

- **name** : *das_string* - Name of the macro

- **_module** : *Module*? - Module where the macro is defined

ast::**VariantMacro**

Adapter for the *AstVariantMacro*.

Fields

- **name** : *das_string* - Name of the macro

ast::**ForLoopMacro**

Adapter for the 'AstForLoopMacro'.

Fields

- **name** : *das_string* - Name of the macro

ast::**CaptureMacro**

Adapter for the *AstCaptureMacro*.

Fields

- **name** : *das_string* - Name of the macro

ast::**TypeMacro**

Compilation time only structure which holds live information about type macro.

Fields

- **name** : *das_string* - Name of the macro

ast::**SimulateMacro**

Adapter for the *AstSimulateMacro*.

Fields

- **name** : *das_string* - Name of the macro.

ast::**ExprReader**

Compilation time only expression which holds temporary information for the *AstReaderMacro*.

Fields

- **at** : *LineInfo* - Location of the expression in source code
- **_type** : smart_ptr< *TypeDecl*> - Type of the expression
- **__rtti** : string - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **genFlags** : *ExprGenFlags* - Expression generation flags
- **flags** : *ExprFlags* - Expression flags
- **printFlags** : *ExprPrintFlags* - Expression print flags
- **macro** : smart_ptr< *ReaderMacro*> - Macro which is attached to the context parser.
- **sequence** : *das_string* - Sequence of characters being read.

ast::**ExprCallMacro**

Fields

- **at** : *LineInfo* - Compilation time only expression which holds temporary information for the *AstCallMacro*.
- **_type** : smart_ptr< *TypeDecl*> - Location of the expression in source code
- **__rtti** : string - Type of the expression
- **genFlags** : *ExprGenFlags* - Runtime type information of the class of the expression (i.e "ExprConstant", "ExprCall", etc)
- **flags** : *ExprFlags* - Expression generation flags
- **printFlags** : *ExprPrintFlags* - Expression flags
- **name** : *das_string* - Expression print flags
- **arguments** : vector<smart_ptr<Expression>> - Name of the macro being called
- **argumentsFailedToInfer** : bool - List of argument expressions
- **atEnclosure** : *LineInfo* - If the arguments failed to infer their types
- **inFunction** : *Function*? - Location of the expression in source code
- **macro** : *CallMacro*? - Call macro, if resolved

10.2.4 Call macros

ast::**quote**

Returns the AST expression tree of the provided code without evaluating or type-inferring it. Used in macro programming to capture source code as a manipulable AST.

10.2.5 Typeinfo macros

ast::**ast_typedecl**

Returns a *TypeDeclPtr* for the type specified via *type<>* or subexpression type, for example *typeinfo(ast_typedecl type<int?>)*. Useful in macros that need compile-time access to type declarations.

ast::**ast_function**

Returns a *FunctionPtr* to the function specified by the subexpression, for example `typeinfo(ast_function @@foo)`. Useful in macros that need compile-time access to function declarations.

10.2.6 Handled types

ast::**MakeStruct**

Annotation representing a vector of *MakeFieldDecl* used to initialize fields in *ExprMakeStruct* expressions.

10.2.7 Classes

`ast::AstFunctionAnnotation`

Annotation macro that attaches to *Function* declarations. Provides compile-time hooks for transforming functions, adding finalization logic, and controlling function compilation behavior.

`ast::AstBlockAnnotation`

Annotation macro that attaches to *ExprBlock* nodes. Provides compile-time hooks for inspecting and transforming code blocks, including loop bodies and scoped blocks.

`ast::AstStructureAnnotation`

Annotation macro that attaches to *Structure* declarations. Provides compile-time hooks for inspecting and modifying structure definitions, fields, and layout.

`ast::AstPassMacro`

Macro that executes as an additional inference pass during compilation. Allows injecting custom analysis and transformation logic into the type-inference pipeline.

`ast::AstVariantMacro`

Macro for implementing custom `is`, `as`, and `?as` expressions. Allows user-defined variant-like dispatch and type-checking patterns beyond the built-in variant type.

`ast::AstForLoopMacro`

Macro for implementing custom for-loop iteration patterns. Intercepts for-loop expressions during compilation, similar to the `visitExprFor` callback of *AstVisitor*.

`ast::AstCaptureMacro`

Macro for implementing custom lambda capture behavior. Controls how variables are captured from the enclosing scope when creating lambdas and generators.

`ast::AstTypeMacro`

Macro that participates in type declarations, enabling syntax like `$macro_name<type_args...>(args)` for custom type construction and transformation.

`ast::AstSimulateMacro`

Macro that hooks into the context simulation phase — the final compilation step where the AST is translated into executable simulation nodes.

`ast::AstReaderMacro`

Macro for implementing custom parsing syntax using the `%MacroName~` notation. The reader macro controls the start and end of the custom parsing region and produces an AST expression from the parsed content.

`ast::AstCommentReader`

Macro for implementing custom comment parsing, such as extracting doxygen-style documentation or other structured metadata from source comments during compilation.

`ast::AstCallMacro`

Macro for implementing custom call-like expressions (e.g. `foo(bar, bar2, ...)`). The macro intercepts specific function calls during compilation and can rewrite them into arbitrary AST.

`ast::AstTypeInfoMacro`

Macro for implementing custom `typeinfo` traits, enabling expressions like `typeinfo(YourTraitHere ...)` that extract compile-time type information.

`ast::AstEnumerationAnnotation`

Annotation macro that attaches to *Enumeration* declarations. Provides compile-time hooks for inspecting and modifying enumeration definitions.

`ast::AstVisitor`

Implements the *Visitor* interface for traversing and transforming the AST tree. Provides `visit` and `preVisit` callbacks for every expression and declaration node type.

10.2.8 Call generation

- *make_call (at: LineInfo; name: string) : smart_ptr<Expression>*

`ast::make_call(at: LineInfo; name: string) : smart_ptr<Expression>()`

Creates the appropriate call expression for a given function name in the program.

Arguments

- **at** : *LineInfo* implicit
- **name** : string implicit

10.2.9 Visitor pattern

- *visit (expression: smart_ptr<TypeDecl>; adapter: smart_ptr<VisitorAdapter>) : smart_ptr<TypeDecl>*
- *visit (program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>)*
- *visit (function: smart_ptr<Function>; adapter: smart_ptr<VisitorAdapter>)*
- *visit (expression: smart_ptr<Expression>; adapter: smart_ptr<VisitorAdapter>) : smart_ptr<Expression>*
- *visit_enumeration (program: smart_ptr<Program>; enumeration: smart_ptr<Enumeration>; adapter: smart_ptr<VisitorAdapter>)*
- *visit_finally (expression: smart_ptr<ExprBlock>; adapter: smart_ptr<VisitorAdapter>)*
- *visit_module (program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>; module: Module?)*
- *visit_modules (program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>)*
- *visit_structure (program: smart_ptr<Program>; structure: smart_ptr<Structure>; adapter: smart_ptr<VisitorAdapter>)*

visit

`ast::visit(expression: smart_ptr<TypeDecl>; adapter: smart_ptr<VisitorAdapter>) : smart_ptr<TypeDecl>()`

Invokes an AST visitor on the given object.

Arguments

- **expression** : smart_ptr< *TypeDecl*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit

`ast::visit`(*program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>*)

`ast::visit`(*function: smart_ptr<Function>; adapter: smart_ptr<VisitorAdapter>*)

`ast::visit(expression: smart_ptr<Expression>; adapter: smart_ptr<VisitorAdapter>) : smart_ptr<Expressio`

`ast::visit_enumeration`(*program: smart_ptr<Program>; enumeration: smart_ptr<Enumeration>; adapter: smart_ptr<VisitorAdapter>*)

Invokes an AST visitor on the given enumeration.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **enumeration** : smart_ptr< *Enumeration*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit

`ast::visit_finally`(*expression: smart_ptr<ExprBlock>; adapter: smart_ptr<VisitorAdapter>*)

Invokes the visitor on the finally section of a block.

Arguments

- **expression** : smart_ptr< *ExprBlock*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit

`ast::visit_module`(*program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>; module: Module?*)

Invokes an AST visitor on the given module.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit
- **module** : *Module*? implicit

`ast::visit_modules`(*program: smart_ptr<Program>; adapter: smart_ptr<VisitorAdapter>*)

Invokes an AST visitor on all modules in the specified program.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit

ast::**visit_structure**(*program: smart_ptr<Program>; structure: smart_ptr<Structure>; adapter: smart_ptr<VisitorAdapter>*)

Invokes an AST visitor on the given structure.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **structure** : smart_ptr< *Structure*> implicit
- **adapter** : smart_ptr< *VisitorAdapter*> implicit

10.2.10 Expression generation

- *force_generated (function: smart_ptr<Function> const&; value: bool)*
- *force_generated (expression: smart_ptr<Expression> const&; value: bool)*
- *get_expression_annotation (expr: Expression?) : Annotation?*
- *make_type_info_structure (ctx: Context; type: smart_ptr<TypeDecl>) : TypeInfo?*

force_generated

ast::**force_generated**(*function: smart_ptr<Function> const&; value: bool*)

Sets the generated flag on an expression and its subexpressions.

Arguments

- **function** : smart_ptr< *Function*>& implicit
- **value** : bool

ast::**force_generated**(*expression: smart_ptr<Expression> const&; value: bool*)

---

ast::**get_expression_annotation(expr: Expression?) : Annotation?**()

Returns the Annotation associated with an Expression or its inherited types.

Arguments

- **expr** : *Expression*? implicit

ast::**make_type_info_structure(ctx: Context; type: smart_ptr<TypeDecl>) : TypeInfo?**()

Returns a new TypeInfo corresponding to the specified type.

Arguments

- **ctx** : *Context* implicit
- **type** : smart_ptr< *TypeDecl*> implicit

10.2.11 Adapter generation

- *make_block_annotation (name: string; var someClassPtr: auto) : FunctionAnnotationPtr*
- *make_block_annotation (name: string; class: void?; info: StructInfo const?) : smart_ptr<FunctionAnnotation>*
- *make_block_type (blk: ExprBlock?) : smart_ptr<TypeDecl>*
- *make_call_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<CallMacro>*
- *make_call_macro (name: string; var someClassPtr: auto) : CallMacroPtr*
- *make_capture_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<CaptureMacro>*
- *make_capture_macro (name: string; var someClassPtr: auto) : CaptureMacroPtr*
- *make_clone_structure (structure: Structure?) : smart_ptr<Function>*
- *make_comment_reader (class: void?; info: StructInfo const?) : smart_ptr<CommentReader>*
- *make_comment_reader (name: string; var someClassPtr: auto) : CommentReaderPtr*
- *make_enum_debug_info (helper: smart_ptr<DebugInfoHelper>; en: Enumeration const?) : EnumInfo?*
- *make_enumeration_annotation (name: string; var someClassPtr: auto) : EnumerationAnnotationPtr*
- *make_enumeration_annotation (name: string; class: void?; info: StructInfo const?) : smart_ptr<EnumerationAnnotation>*
- *make_for_loop_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<ForLoopMacro>*
- *make_for_loop_macro (name: string; var someClassPtr: auto) : ForLoopMacroPtr*
- *make_function_annotation (name: string; var someClassPtr: auto) : FunctionAnnotationPtr*
- *make_function_annotation (name: string; class: void?; info: StructInfo const?) : smart_ptr<FunctionAnnotation>*
- *make_function_debug_info (helper: smart_ptr<DebugInfoHelper>; fn: Function const?) : FuncInfo?*
- *make_invokable_type_debug_info (helper: smart_ptr<DebugInfoHelper>; blk: smart_ptr<TypeDecl>; at: LineInfo) : FuncInfo?*
- *make_pass_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<PassMacro>*
- *make_pass_macro (name: string; var someClassPtr: auto) : PassMacroPtr*
- *make_reader_macro (name: string; var someClassPtr: auto) : ReaderMacroPtr*
- *make_reader_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<ReaderMacro>*
- *make_simulate_macro (name: string; var someClassPtr: auto) : SimulateMacroPtr*
- *make_simulate_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<SimulateMacro>*
- *make_struct_debug_info (helper: smart_ptr<DebugInfoHelper>; st: Structure const?) : StructInfo?*
- *make_struct_variable_debug_info (helper: smart_ptr<DebugInfoHelper>; st: Structure const?; var: FieldDeclaration const?) : VarInfo?*
- *make_structure_annotation (name: string; var someClassPtr: auto) : StructureAnnotationPtr*
- *make_structure_annotation (name: string; class: void?; info: StructInfo const?) : smart_ptr<StructureAnnotation>*
- *make_type_info (helper: smart_ptr<DebugInfoHelper>; info: TypeInfo?; type: smart_ptr<TypeDecl> const&) : TypeInfo?*

- *make_type_macro (name: string; var someClassPtr: auto) : TypeMacroPtr*
- *make_type_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<TypeMacro>*
- *make_typeinfo_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<TypeInfoMacro>*
- *make_typeinfo_macro (name: string; var someClassPtr: auto) : TypeInfoMacroPtr*
- *make_variable_debug_info (helper: smart_ptr<DebugInfoHelper>; var: Variable?) : VarInfo?*
- *make_variant_macro (name: string; var someClassPtr: auto) : VariantMacroPtr*
- *make_variant_macro (name: string; class: void?; info: StructInfo const?) : smart_ptr<VariantMacro>*
- *make_visitor (someClass: auto) : smart_ptr<VisitorAdapter>*
- *make_visitor (class: void?; info: StructInfo const?) : smart_ptr<VisitorAdapter>*

make_block_annotation

`ast::make_block_annotation(name: string; someClassPtr: auto) : FunctionAnnotationPtr()`

Creates an adapter for the AstBlockAnnotation interface.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::make_block_annotation(name: string; class: void?; info: StructInfo const?) : smart_ptr<FunctionAnn`

`ast::make_block_type(blk: ExprBlock?) : smart_ptr<TypeDecl>()`

Generates a TypeDeclPtr for a specified block or lambda type.

Arguments

- **blk** : *ExprBlock*? implicit

make_call_macro

`ast::make_call_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<CallMacro>()`

Creates an adapter for the AstCallMacro interface.

Arguments

- **name** : string implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

`ast::make_call_macro(name: string; someClassPtr: auto) : CallMacroPtr()`

---

make_capture_macro

`ast::make_capture_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<CaptureMacro>(`

Creates an adapter for the AstCaptureMacro interface.

Arguments

- **name** : string implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

`ast::make_capture_macro(name: string; someClassPtr: auto) : CaptureMacroPtr()`

`ast::make_clone_structure(structure: Structure?) : smart_ptr<Function>()`

Generates a clone function for the given structure.

Arguments

- **structure** : *Structure*? implicit

make_comment_reader

`ast::make_comment_reader(class: void?; info: StructInfo const?) : smart_ptr<CommentReader>()`

Creates an adapter for the AstCommentReader interface.

Arguments

- **class** : void? implicit
- **info** : *StructInfo*? implicit

`ast::make_comment_reader(name: string; someClassPtr: auto) : CommentReaderPtr()`

`ast::make_enum_debug_info(helper: smart_ptr<DebugInfoHelper>; en: Enumeration const?) : EnumInfo?()`

Generates an EnumInfo for the specified enumeration using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **en** : *Enumeration*? implicit

make_enumeration_annotation

`ast::make_enumeration_annotation(name: string; someClassPtr: auto) : EnumerationAnnotationPtr()`

Creates an adapter for the AstEnumerationAnnotation interface.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::make_enumeration_annotation(name: string; class: void?; info: StructInfo const?) : smart_ptr<Enume`

make_for_loop_macro

`ast::make_for_loop_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<ForLoopMacro>`

Creates an adapter for the AstForLoopMacro interface.

Arguments

- **name** : string implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

`ast::make_for_loop_macro(name: string; someClassPtr: auto) : ForLoopMacroPtr()`

make_function_annotation

`ast::make_function_annotation(name: string; someClassPtr: auto) : FunctionAnnotationPtr()`

Creates an adapter for the AstFunctionAnnotation interface.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::make_function_annotation(name: string; class: void?; info: StructInfo const?) : smart_ptr<Function`

`ast::make_function_debug_info(helper: smart_ptr<DebugInfoHelper>; fn: Function const?) : FuncInfo?()`

Generates a FuncInfo for the specified function using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **fn** : *Function*? implicit

`ast::make_invokable_type_debug_info(helper: smart_ptr<DebugInfoHelper>; blk: smart_ptr<TypeDecl>; at: L`

Generates a FuncInfo for an invokable type such as a lambda or block using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **blk** : smart_ptr< *TypeDecl*> implicit
- **at** : *LineInfo* implicit

make_pass_macro

ast::**make_pass_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<PassMacro>**()

Creates an adapter for the AstPassMacro interface.

Arguments

- **name** : string implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

ast::**make_pass_macro(name: string; someClassPtr: auto) : PassMacroPtr**()

---

make_reader_macro

ast::**make_reader_macro(name: string; someClassPtr: auto) : ReaderMacroPtr**()

Creates an adapter for the AstReaderMacro interface.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**make_reader_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<ReaderMacro>**()

---

make_simulate_macro

ast::**make_simulate_macro(name: string; someClassPtr: auto) : SimulateMacroPtr**()

Creates an adapter for the AstSimulateMacro interface.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**make_simulate_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<SimulateMacro**

---

ast::**make_struct_debug_info(helper: smart_ptr<DebugInfoHelper>; st: Structure const?) : StructInfo?**()

Generates a StructInfo for the specified structure using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **st** : *Structure*? implicit

ast::**make_struct_variable_debug_info(helper: smart_ptr<DebugInfoHelper>; st: Structure const?; var: Fie**

Generates a VariableInfo for a structure field using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **st** : *Structure*? implicit
- **var** : *FieldDeclaration*? implicit

make_structure_annotation

ast::**make_structure_annotation(name: string; someClassPtr: auto) : StructureAnnotationPtr**()

Creates an adapter for the AstStructureAnnotation interface.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**make_structure_annotation(name: string; class: void?; info: StructInfo const?) : smart_ptr<Structu**

---

ast::**make_type_info(helper: smart_ptr<DebugInfoHelper>; info: TypeInfo?; type: smart_ptr<TypeDecl> cons**

Generates a TypeInfo for the specified type using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **info** : *TypeInfo*? implicit
- **type** : smart_ptr< *TypeDecl*>& implicit

make_type_macro

ast::**make_type_macro(name: string; someClassPtr: auto) : TypeMacroPtr**()

Creates an adapter for the AstTypeMacro interface.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**make_type_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<TypeMacro>**()

---

make_typeinfo_macro

ast::**make_typeinfo_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<TypeInfoMacro**

Creates an adapter for the AstTypeInfoMacro interface.

Arguments

- **name** : string implicit
- **class** : void? implicit
- **info** : *StructInfo*? implicit

ast::**make_typeinfo_macro(name: string; someClassPtr: auto) : TypeInfoMacroPtr**()

---

ast::**make_variable_debug_info(helper: smart_ptr<DebugInfoHelper>; var: Variable?) : VarInfo?**()

Generates a VariableInfo for the specified variable using the given DebugInfoHelper.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **var** : *Variable*? implicit

make_variant_macro

ast::**make_variant_macro(name: string; someClassPtr: auto) : VariantMacroPtr**()

Creates an adapter for the AstVariantMacro interface.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**make_variant_macro(name: string; class: void?; info: StructInfo const?) : smart_ptr<VariantMacro>**(

---

make_visitor

ast::**make_visitor(someClass: auto) : smart_ptr<VisitorAdapter>**()

Creates an adapter for the AstVisitor interface.

Arguments

- **someClass** : auto

ast::**make_visitor(class: void?; info: StructInfo const?) : smart_ptr<VisitorAdapter>**()

10.2.12 Adapter application

- *add_block_annotation (block: smart_ptr<ExprBlock>; annotation: smart_ptr<FunctionAnnotation>&)*
- *add_block_annotation (block: smart_ptr<ExprBlock>; annotation: smart_ptr<AnnotationDeclaration>&)*
- *add_call_macro (module: Module?; annotation: smart_ptr<CallMacro>&)*
- *add_capture_macro (module: Module?; annotation: smart_ptr<CaptureMacro>&)*
- *add_comment_reader (module: Module?; reader: smart_ptr<CommentReader>&)*
- *add_dirty_infer_macro (module: Module?; annotation: smart_ptr<PassMacro>&)*
- *add_enumeration_annotation (module: Module?; annotation: smart_ptr<EnumerationAnnotation>&)*
- *add_for_loop_macro (module: Module?; annotation: smart_ptr<ForLoopMacro>&)*
- *add_function_annotation (function: smart_ptr<Function>; annotation: smart_ptr<FunctionAnnotation>&)*
- *add_function_annotation (module: Module?; annotation: smart_ptr<FunctionAnnotation>&)*
- *add_function_annotation (function: smart_ptr<Function>; annotation: smart_ptr<AnnotationDeclaration>&)*
- *add_global_lint_macro (module: Module?; annotation: smart_ptr<PassMacro>&)*
- *add_infer_macro (module: Module?; annotation: smart_ptr<PassMacro>&)*
- *add_lint_macro (module: Module?; annotation: smart_ptr<PassMacro>&)*
- *add_module_option (module: Module?; option: string; type: Type)*
- *add_new_block_annotation (name: string; var someClassPtr: auto) : auto*
- *add_new_call_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_capture_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_comment_reader (name: string; var someClassPtr: auto) : auto*
- *add_new_contract_annotation (name: string; var someClassPtr: auto) : auto*
- *add_new_dirty_infer_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_enumeration_annotation (name: string; var someClassPtr: auto) : auto*
- *add_new_for_loop_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_function_annotation (name: string; var someClassPtr: auto) : auto*
- *add_new_global_lint_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_infer_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_lint_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_optimization_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_reader_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_simulate_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_structure_annotation (name: string; var someClassPtr: auto) : auto*
- *add_new_type_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_typeinfo_macro (name: string; var someClassPtr: auto) : auto*
- *add_new_variant_macro (name: string; var someClassPtr: auto) : auto*
- *add_optimization_macro (module: Module?; annotation: smart_ptr<PassMacro>&)*

- *add_reader_macro (module: Module?; annotation: smart_ptr<ReaderMacro>&)*
- *add_simulate_macro (module: Module?; annotation: smart_ptr<SimulateMacro>&)*
- *add_structure_annotation (structure: smart_ptr<Structure>; annotation: smart_ptr<StructureAnnotation>&)*
- *add_structure_annotation (module: Module?; annotation: smart_ptr<StructureAnnotation>&)*
- *add_structure_annotation (structure: smart_ptr<Structure>; annotation: smart_ptr<AnnotationDeclaration>&)*
- *add_type_macro (module: Module?; annotation: smart_ptr<TypeMacro>&)*
- *add_typeinfo_macro (module: Module?; annotation: smart_ptr<TypeInfoMacro>&)*
- *add_variant_macro (module: Module?; annotation: smart_ptr<VariantMacro>&)*

add_block_annotation

`ast::`**`add_block_annotation`**(*block: smart_ptr<ExprBlock>; annotation: smart_ptr<FunctionAnnotation>&*)

Adds an annotation declaration to a block.

Arguments

- **block** : smart_ptr< *ExprBlock*> implicit
- **annotation** : smart_ptr< *FunctionAnnotation*>& implicit

`ast::`**`add_block_annotation`**(*block: smart_ptr<ExprBlock>; annotation: smart_ptr<AnnotationDeclaration>&*)

`ast::`**`add_call_macro`**(*module: Module?; annotation: smart_ptr<CallMacro>&*)

Adds an AstCallMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *CallMacro*>& implicit

`ast::`**`add_capture_macro`**(*module: Module?; annotation: smart_ptr<CaptureMacro>&*)

Adds an AstCaptureMacro to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *CaptureMacro*>& implicit

`ast::`**`add_comment_reader`**(*module: Module?; reader: smart_ptr<CommentReader>&*)

Adds an AstCommentReader adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **reader** : smart_ptr< *CommentReader*>& implicit

`ast::`**`add_dirty_infer_macro`**`(`*module: Module?; annotation: smart_ptr<PassMacro>&*`)`

Adds an AstPassMacro adapter to the dirty inference pass.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *PassMacro*>& implicit

`ast::`**`add_enumeration_annotation`**`(`*module: Module?; annotation: smart_ptr<EnumerationAnnotation>&*`)`

Adds an annotation to an enumeration and calls apply if applicable.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *EnumerationAnnotation*>& implicit

`ast::`**`add_for_loop_macro`**`(`*module: Module?; annotation: smart_ptr<ForLoopMacro>&*`)`

Adds an AstForLoopMacro to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *ForLoopMacro*>& implicit

add_function_annotation

`ast::`**`add_function_annotation`**`(`*function: smart_ptr<Function>; annotation: smart_ptr<FunctionAnnotation>&*`)`

Adds an annotation to a function and calls apply if applicable.

Arguments

- **function** : smart_ptr< *Function*> implicit
- **annotation** : smart_ptr< *FunctionAnnotation*>& implicit

`ast::`**`add_function_annotation`**`(`*module: Module?; annotation: smart_ptr<FunctionAnnotation>&*`)`

`ast::`**`add_function_annotation`**`(`*function: smart_ptr<Function>; annotation: smart_ptr<AnnotationDeclaration>&*`)`

`ast::`**`add_global_lint_macro`**`(`*module: Module?; annotation: smart_ptr<PassMacro>&*`)`

Adds an AstPassMacro adapter to the global lint pass.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *PassMacro*>& implicit

ast::**add_infer_macro**(*module: Module?; annotation: smart_ptr<PassMacro>&*)

Adds an AstPassMacro adapter to the type inference pass.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *PassMacro*>& implicit

ast::**add_lint_macro**(*module: Module?; annotation: smart_ptr<PassMacro>&*)

Adds an AstPassMacro adapter to the lint pass of the current module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *PassMacro*>& implicit

ast::**add_module_option**(*module: Module?; option: string; type: Type*)

Adds a module-specific option accessible via the *options* keyword.

Arguments

- **module** : *Module*? implicit
- **option** : string implicit
- **type** : *Type*

ast::**add_new_block_annotation(name: string; someClassPtr: auto) : auto**()

Creates an AstBlockAnnotation adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**add_new_call_macro(name: string; someClassPtr: auto) : auto**()

Creates an AstCallMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**add_new_capture_macro(name: string; someClassPtr: auto) : auto**()

Creates an AstCaptureMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**add_new_comment_reader(name: string; someClassPtr: auto) : auto**()

Creates an AstCommentReader adapter and adds it to the current module.

Arguments

- **name** : string

- **someClassPtr** : auto

`ast::add_new_contract_annotation(name: string; someClassPtr: auto) : auto()`

Creates an AstContractAnnotation adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_dirty_infer_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstPassMacro adapter and adds it to the current module's dirty infer pass.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_enumeration_annotation(name: string; someClassPtr: auto) : auto()`

Creates an AstEnumerationAnnotation adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_for_loop_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstForLoopMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_function_annotation(name: string; someClassPtr: auto) : auto()`

Creates an AstFunctionAnnotation adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_global_lint_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstPassMacro adapter and adds it to the current module's global lint pass.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_infer_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstPassMacro adapter and adds it to the current module's infer pass.

Arguments

- **name** : string

- **someClassPtr** : auto

`ast::add_new_lint_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstPassMacro adapter and adds it to the current module's lint pass.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_optimization_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstPassMacro adapter and adds it to the current module's optimization pass.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_reader_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstReaderMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_simulate_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstSimulateMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_structure_annotation(name: string; someClassPtr: auto) : auto()`

Creates an AstStructureAnnotation adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_type_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstTypeMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

`ast::add_new_typeinfo_macro(name: string; someClassPtr: auto) : auto()`

Creates an AstTypeInfoMacro adapter and adds it to the current module.

Arguments

- **name** : string

- **someClassPtr** : auto

ast::**add_new_variant_macro(name: string; someClassPtr: auto) : auto**()

Creates an AstVariantMacro adapter and adds it to the current module.

Arguments

- **name** : string
- **someClassPtr** : auto

ast::**add_optimization_macro**(*module: Module?; annotation: smart_ptr<PassMacro>&*)

Adds an AstPassMacro adapter to the optimization pass of a specific module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *PassMacro*>& implicit

ast::**add_reader_macro**(*module: Module?; annotation: smart_ptr<ReaderMacro>&*)

Adds an AstReaderMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *ReaderMacro*>& implicit

ast::**add_simulate_macro**(*module: Module?; annotation: smart_ptr<SimulateMacro>&*)

Adds an AstSimulateMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *SimulateMacro*>& implicit

add_structure_annotation

ast::**add_structure_annotation**(*structure: smart_ptr<Structure>; annotation: smart_ptr<StructureAnnotation>&*)

Adds a structure annotation to the given object, calling apply if applicable.

Arguments

- **structure** : smart_ptr< *Structure*> implicit
- **annotation** : smart_ptr< *StructureAnnotation*>& implicit

ast::**add_structure_annotation**(*module: Module?; annotation: smart_ptr<StructureAnnotation>&*)

ast::**add_structure_annotation**(*structure: smart_ptr<Structure>; annotation: smart_ptr<AnnotationDeclaration>&*)

ast::**add_type_macro**(*module: Module?; annotation: smart_ptr<TypeMacro>&*)

Adds an AstTypeMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *TypeMacro*>& implicit

ast::**add_typeinfo_macro**(*module: Module?; annotation: smart_ptr<TypeInfoMacro>&*)

Adds an AstTypeInfoMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *TypeInfoMacro*>& implicit

ast::**add_variant_macro**(*module: Module?; annotation: smart_ptr<VariantMacro>&*)

Adds an AstVariantMacro adapter to the specified module.

Arguments

- **module** : *Module*? implicit
- **annotation** : smart_ptr< *VariantMacro*>& implicit

10.2.13 Adding objects to objects

- *add_alias (module: Module?; structure: smart_ptr<TypeDecl>&) : bool*
- *add_enumeration_entry (enum: smart_ptr<Enumeration>; name: string) : int*
- *add_function (module: Module?; function: smart_ptr<Function>&) : bool*
- *add_generic (module: Module?; function: smart_ptr<Function>&) : bool*
- *add_keyword (module: Module?; keyword: string; needOxfordComma: bool) : bool*
- *add_module_require (module: Module?; publicModule: Module?; pub: bool) : bool*
- *add_structure (module: Module?; structure: smart_ptr<Structure>&) : bool*
- *add_structure_alias (structure: Structure?; aliasName: string; alias: smart_ptr<TypeDecl> const&) : bool*
- *add_type_function (module: Module?; keyword: string) : bool*
- *add_variable (module: Module?; variable: smart_ptr<Variable>&) : bool*

ast::**add_alias(module: Module?; structure: smart_ptr<TypeDecl>&) : bool**()

Adds a type alias to the specified module.

Arguments

- **module** : *Module*? implicit
- **structure** : smart_ptr< *TypeDecl*>& implicit

ast::**add_enumeration_entry(enum: smart_ptr<Enumeration>; name: string) : int**()

Adds a new entry to an enumeration annotation.

Arguments

- **enum** : smart_ptr< *Enumeration*> implicit
- **name** : string implicit

`ast::add_function(module: Module?; function: smart_ptr<Function>&) : bool()`

Adds a function to a module, returning false if a duplicate already exists.

Arguments

- **module** : *Module*? implicit
- **function** : smart_ptr< *Function*>& implicit

`ast::add_generic(module: Module?; function: smart_ptr<Function>&) : bool()`

Adds a generic function to a module, returning false if a duplicate already exists.

Arguments

- **module** : *Module*? implicit
- **function** : smart_ptr< *Function*>& implicit

`ast::add_keyword(module: Module?; keyword: string; needOxfordComma: bool) : bool()`

Registers a new keyword in the specified module, making it available to the parser.

Arguments

- **module** : *Module*? implicit
- **keyword** : string implicit
- **needOxfordComma** : bool

`ast::add_module_require(module: Module?; publicModule: Module?; pub: bool) : bool()`

Adds module dependencies, similar to the *require* keyword.

Arguments

- **module** : *Module*? implicit
- **publicModule** : *Module*? implicit
- **pub** : bool

`ast::add_structure(module: Module?; structure: smart_ptr<Structure>&) : bool()`

Adds a structure to a module, returning false if a duplicate already exists.

Arguments

- **module** : *Module*? implicit
- **structure** : smart_ptr< *Structure*>& implicit

`ast::add_structure_alias(structure: Structure?; aliasName: string; alias: smart_ptr<TypeDecl> const&) :`

Adds a typedef alias to a structure type in the AST, equivalent to a typedef in the structure body.

Arguments

- **structure** : *Structure*? implicit
- **aliasName** : string implicit
- **alias** : smart_ptr< *TypeDecl*>& implicit

`ast::`**`add_type_function(module: Module?; keyword: string) : bool`**`()`

Adds a type function keyword, allowing function calls to accept type arguments before regular arguments via the *some_call<type_args>(regular_args)* syntax.

Arguments

- **module** : *Module*? implicit
- **keyword** : string implicit

`ast::`**`add_variable(module: Module?; variable: smart_ptr<Variable>&) : bool`**`()`

Adds a variable to a module, returning false if a duplicate already exists.

Arguments

- **module** : *Module*? implicit
- **variable** : smart_ptr< *Variable*>& implicit

10.2.14 Program and module access

- *compiling_module () : Module?*
- *compiling_program () : smart_ptr<Program>*
- *this_module () : Module?*
- *this_program () : smart_ptr<Program>*

`ast::`**`compiling_module() : Module?`**`()`

Returns the module currently being compiled.

`ast::`**`compiling_program() : smart_ptr<Program>`**`()`

Returns the program currently being compiled.

`ast::`**`this_module() : Module?`**`()`

Returns the main module attached to the current context, throwing an error if RTTI is disabled.

`ast::`**`this_program() : smart_ptr<Program>`**`()`

Returns the program attached to the current context, or null if RTTI is disabled.

10.2.15 Textual descriptions of the objects

- *das_to_string (type: Type) : string*
- *describe (expr: smart_ptr<Function>) : auto*
- *describe (expr: smart_ptr<Expression>) : auto*
- *describe (decl: smart_ptr<TypeDecl>; extra: bool = true; contracts: bool = true; modules: bool = true) : auto*
- *describe_cpp (decl: smart_ptr<TypeDecl>; substitureRef: bool = false; skipRef: bool = false; skipConst: bool = false; redundantConst: bool = true; chooseSmartPtr: bool = true) : auto*
- *describe_expression (expression: smart_ptr<Expression>) : string*
- *describe_function (function: smart_ptr<Function>) : string*

- *describe_typedecl (type: smart_ptr<TypeDecl>; extra: bool; contracts: bool; module: bool) : string*
- *describe_typedecl_cpp (type: smart_ptr<TypeDecl>; substitueRef: bool; skipRef: bool; skipConst: bool; redundantConst: bool; choose_smart_ptr: bool) : string*

`ast::das_to_string(type: Type) : string()`

Returns the name of the corresponding daslang base type as a string.

Arguments

- **type** : *Type*

describe

`ast::describe(expr: smart_ptr<Function>) : auto()`

Produces a daslang source code string representation of the given AST object.

Arguments

- **expr** : smart_ptr< *Function*>

`ast::describe(expr: smart_ptr<Expression>) : auto()`

`ast::describe(decl: smart_ptr<TypeDecl>; extra: bool = true; contracts: bool = true; modules: bool = tru`

`ast::describe_cpp(decl: smart_ptr<TypeDecl>; substitureRef: bool = false; skipRef: bool = false; skipCo`

Produces a C++ source code string representation of the given TypeDecl.

Arguments

- **decl** : smart_ptr< *TypeDecl*>
- **substitureRef** : bool
- **skipRef** : bool
- **skipConst** : bool
- **redundantConst** : bool
- **chooseSmartPtr** : bool

`ast::describe_expression(expression: smart_ptr<Expression>) : string()`

Returns a string description of the Expression matching the corresponding daslang source code.

Arguments

- **expression** : smart_ptr< *Expression*> implicit

`ast::describe_function(function: smart_ptr<Function>) : string()`

Returns a string description of the Function matching the corresponding daslang function declaration.

Arguments

- **function** : smart_ptr< *Function*> implicit `ast::`**`describe_typedecl(type: smart_ptr<TypeDecl>; extra: bool; contracts: bool; module: bool) : string(`**

Returns a string description of the TypeDecl matching the corresponding daslang type declaration.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **extra** : bool
- **contracts** : bool
- **module** : bool

`ast::`**`describe_typedecl_cpp(type: smart_ptr<TypeDecl>; substitueRef: bool; skipRef: bool; skipConst: boo`**

Returns a string description of the TypeDecl matching the corresponding C++ type declaration.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **substitueRef** : bool
- **skipRef** : bool
- **skipConst** : bool
- **redundantConst** : bool
- **choose_smart_ptr** : bool

10.2.16 Searching

- *find_bitfield_name (bit: smart_ptr<TypeDecl>; value: bitfield) : string*
- *find_call_macro (module: Module?; name: string) : CallMacro?*
- *find_compiling_function_by_mangled_name_hash (moduleName: string; mangledNameHash: uint64) : smart_ptr<Function>*
- *find_compiling_module (name: string) : Module?*
- *find_enum_name (enum: Enumeration?; value: int64) : string*
- *find_enum_value (enum: Enumeration?; value: string) : int64*
- *find_enum_value (enum: smart_ptr<Enumeration>; value: string) : int64*
- *find_matching_variable (program: Program?; function: Function?; name: string; seePrivate: bool; block: block<(array<smart_ptr<Variable>>#):void>)*
- *find_module (name: string) : Module?*
- *find_module (prog: smart_ptr<Program>; name: string) : Module?*
- *find_module_function_via_rtti (module: Module?; function: function<():void>) : smart_ptr<Function>*
- *find_module_via_rtti (program: smart_ptr<Program>; name: string) : Module?*
- *find_struct_field_parent (structure: smart_ptr<Structure>; name: string) : Structure const?*
- *find_structure_field (structPtr: Structure?; field: string) : FieldDeclaration?*
- *find_unique_structure (program: smart_ptr<Program>; name: string) : Structure?*
- *find_variable (module: Module?; variable: string) : smart_ptr<Variable>*

ast::**find_bitfield_name(bit: smart_ptr<TypeDecl>; value: bitfield) : string**()

Finds the name of a bitfield value in the specified type.

Arguments

- **bit** : smart_ptr< *TypeDecl*> implicit
- **value** : bitfield<>

ast::**find_call_macro(module: Module?; name: string) : CallMacro?**()

Finds a CallMacro by name in the specified module.

Arguments

- **module** : *Module*? implicit
- **name** : string implicit

ast::**find_compiling_function_by_mangled_name_hash(moduleName: string; mangledNameHash: uint64) : smart_**

Returns a Function from the currently compiling program given its mangled name hash.

Arguments

- **moduleName** : string implicit
- **mangledNameHash** : uint64

ast::**find_compiling_module(name: string) : Module?**()

Finds a module by name in the currently compiling program.

Arguments

- **name** : string

ast::**find_enum_name(enum: Enumeration?; value: int64) : string**()

Finds the name corresponding to an enumeration value in the specified type.

Arguments

- **enum** : *Enumeration*? implicit
- **value** : int64

find_enum_value

ast::**find_enum_value(enum: Enumeration?; value: string) : int64**()

Finds the integer value corresponding to an enumeration name in the specified type.

Arguments

- **enum** : *Enumeration*? implicit
- **value** : string implicit

ast::**find_enum_value(enum: smart_ptr<Enumeration>; value: string) : int64**()

ast::**find_matching_variable**(*program: Program?; function: Function?; name: string; seePrivate: bool; block: block<(array<smart_ptr<Variable>>#):void>*)

Finds a global or shared variable accessible from the given function, according to visibility and privacy rules.

Arguments

- **program** : *Program*? implicit
- **function** : *Function*? implicit
- **name** : string implicit
- **seePrivate** : bool
- **block** : block<(array<smart_ptr< *Variable*>>#):void> implicit

find_module

ast::**find_module(name: string) : Module?**()

Finds a module by name in the specified program.

Arguments

- **name** : string

ast::**find_module(prog: smart_ptr<Program>; name: string) : Module?**()

---

ast::**find_module_function_via_rtti(module: Module?; function: function<():void>) : smart_ptr<Function>**(

Finds a function by name in the specified module using RTTI.

Arguments

- **module** : *Module*? implicit
- **function** : function<void>

ast::**find_module_via_rtti(program: smart_ptr<Program>; name: string) : Module?**()

Finds a module by name in the specified program using RTTI.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **name** : string implicit

ast::**find_struct_field_parent(structure: smart_ptr<Structure>; name: string) : Structure const?**()

Finds the parent structure that declares the specified field.

Arguments

- **structure** : smart_ptr< *Structure*> implicit
- **name** : string implicit

ast::**find_structure_field(structPtr: Structure?; field: string) : FieldDeclaration?**()

Returns the FieldDeclaration for a specific field of a structure type, or null if not found.

Arguments

- **structPtr** : *Structure*? implicit
- **field** : string implicit

ast::**find_unique_structure(program: smart_ptr<Program>; name: string) : Structure?**()

Finds a uniquely named structure in the program, returning it if unique or null if ambiguous.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **name** : string implicit

ast::**find_variable(module: Module?; variable: string) : smart_ptr<Variable>**()

Finds a variable by name in the specified module.

Arguments

- **module** : *Module*? implicit
- **variable** : string implicit

10.2.17 Iterating

- *any_array_foreach (array: void?; stride: int; block: block<(void?):void>)*
- *any_table_foreach (table: void?; keyStride: int; valueStride: int; block: block<(void?;void?):void>)*
- *for_each_annotation_ordered (module: Module?; block: block<(uint64;uint64):void>)*
- *for_each_call_macro (module: Module?; block: block<(string#):void>)*
- *for_each_enumeration (module: Module?; block: block<(smart_ptr<Enumeration>):void>)*
- *for_each_field (annotation: BasicStructureAnnotation; block: block<(string;string;smart_ptr<TypeDecl>;uint):void>)*
- *for_each_for_loop_macro (module: Module?; block: block<(smart_ptr<ForLoopMacro>):void>)*
- *for_each_function (module: Module?; name: string; block: block<(smart_ptr<Function>):void>)*
- *for_each_generic (module: Module?; name: string; block: block<(smart_ptr<Function>):void>)*
- *for_each_generic (module: Module?; block: block<(smart_ptr<Function>):void>)*
- *for_each_global (module: Module?; block: block<(smart_ptr<Variable>):void>)*
- *for_each_module (program: Program?; block: block<(Module?):void>)*
- *for_each_module_function (module: Module?; blk: block<(smart_ptr<Function>):void>)*
- *for_each_module_no_order (program: Program?; block: block<(Module?):void>)*
- *for_each_reader_macro (module: Module?; block: block<(string#):void>)*
- *for_each_structure (module: Module?; block: block<(smart_ptr<Structure>):void>)*
- *for_each_structure_alias (structure: Structure?; block: block<(smart_ptr<TypeDecl>):void>)*
- *for_each_typedef (module: Module?; block: block<(string#;smart_ptr<TypeDecl>):void>)*

- *for_each_typeinfo_macro (module: Module?; block: block<(smart_ptr<TypeInfoMacro>):void>)*
- *for_each_typemacro (module: Module?; block: block<(smart_ptr<TypeMacro>):void>)*
- *for_each_variant_macro (module: Module?; block: block<(smart_ptr<VariantMacro>):void>)*

ast::**any_array_foreach**(*array: void?; stride: int; block: block<(void?):void>*)

Iterates through any *array*<> type in a typeless fashion using *void?* pointers.

Arguments

- **array** : void? implicit
- **stride** : int
- **block** : block<(void?):void> implicit

ast::**any_table_foreach**(*table: void?; keyStride: int; valueStride: int; block: block<(void?;void?):void>*)

Iterates through any *table*<> type in a typeless fashion using *void?* pointers.

Arguments

- **table** : void? implicit
- **keyStride** : int
- **valueStride** : int
- **block** : block<(void?;void?):void> implicit

ast::**for_each_annotation_ordered**(*module: Module?; block: block<(uint64;uint64):void>*)

Iterates through each annotation in the given module in the order they were added.

Arguments

- **module** : *Module*? implicit
- **block** : block<(uint64;uint64):void> implicit

ast::**for_each_call_macro**(*module: Module?; block: block<(string#):void>*)

Iterates through every CallMacro adapter in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(string#):void> implicit

ast::**for_each_enumeration**(*module: Module?; block: block<(smart_ptr<Enumeration>):void>*)

Iterates through every enumeration in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *Enumeration*>):void> implicit

ast::**for_each_field**(*annotation: BasicStructureAnnotation; block: block<(string;string;smart_ptr<TypeDecl>;uint):void>*)

Iterates through every field in a BuiltinStructure handled type.

Arguments

- **annotation** : *BasicStructureAnnotation* implicit
- **block** : block<(string;string;smart_ptr< *TypeDecl*>;uint):void> implicit

`ast::`**`for_each_for_loop_macro`**`(`*module: Module?; block: block<(smart_ptr<ForLoopMacro>):void>*`)`

Iterates through every for-loop macro in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *ForLoopMacro*>):void> implicit

`ast::`**`for_each_function`**`(`*module: Module?; name: string; block: block<(smart_ptr<Function>):void>*`)`

Iterates through each function in the given module, matching all functions if the name is empty.

Arguments

- **module** : *Module*? implicit
- **name** : string implicit
- **block** : block<(smart_ptr< *Function*>):void> implicit

for_each_generic

`ast::`**`for_each_generic`**`(`*module: Module?; name: string; block: block<(smart_ptr<Function>):void>*`)`

Iterates through each generic function in the given module.

Arguments

- **module** : *Module*? implicit
- **name** : string implicit
- **block** : block<(smart_ptr< *Function*>):void> implicit

`ast::`**`for_each_generic`**`(`*module: Module?; block: block<(smart_ptr<Function>):void>*`)`

`ast::`**`for_each_global`**`(`*module: Module?; block: block<(smart_ptr<Variable>):void>*`)`

Iterates through every global variable in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *Variable*>):void> implicit

`ast::`**`for_each_module`**`(`*program: Program?; block: block<(Module?):void>*`)`

Iterates through each module in the program in dependency order.

Arguments

- **program** : *Program*? implicit
- **block** : block<(*Module*?):void> implicit

ast::**for_each_module_function**(*module: Module?; blk: block<(smart_ptr<Function>):void>*)

Iterates through each function in the given module.

Arguments

- **module** : *Module*? implicit
- **blk** : block<(smart_ptr< *Function*>):void> implicit

ast::**for_each_module_no_order**(*program: Program?; block: block<(Module?):void>*)

Iterates through each module in the program in no particular order, as they appear in the library group.

Arguments

- **program** : *Program*? implicit
- **block** : block<(*Module*?):void> implicit

ast::**for_each_reader_macro**(*module: Module?; block: block<(string#):void>*)

Iterates through each reader macro in the given module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(string#):void> implicit

ast::**for_each_structure**(*module: Module?; block: block<(smart_ptr<Structure>):void>*)

Iterates through every structure in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *Structure*>):void> implicit

ast::**for_each_structure_alias**(*structure: Structure?; block: block<(smart_ptr<TypeDecl>):void>*)

Iterates over all structure aliases defined in the given structure type, invoking the provided block for each alias.

Arguments

- **structure** : *Structure*? implicit
- **block** : block<(smart_ptr< *TypeDecl*>):void> implicit

ast::**for_each_typedef**(*module: Module?; block: block<(string#;smart_ptr<TypeDecl>):void>*)

Iterates through every typedef in the specified module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(string#;smart_ptr< *TypeDecl*>):void> implicit

ast::**for_each_typeinfo_macro**(*module: Module?; block: block<(smart_ptr<TypeInfoMacro>):void>*)

Iterates through each typeinfo macro in the given module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *TypeInfoMacro*>):void> implicit

ast::**for_each_typemacro**(*module: Module?; block: block<(smart_ptr<TypeMacro>):void>*)

Iterates over all type macros registered in the given module, invoking the provided block for each one.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *TypeMacro*>):void> implicit

ast::**for_each_variant_macro**(*module: Module?; block: block<(smart_ptr<VariantMacro>):void>*)

Iterates through each variant macro in the given module.

Arguments

- **module** : *Module*? implicit
- **block** : block<(smart_ptr< *VariantMacro*>):void> implicit

10.2.18 Cloning

- *clone_expression (expression: smart_ptr<Expression>) : smart_ptr<Expression>*
- *clone_file_info (name: string; tab_size: int) : FileInfo?*
- *clone_function (function: smart_ptr<Function>) : smart_ptr<Function>*
- *clone_function (fn: Function?) : FunctionPtr*
- *clone_structure (structure: Structure const?) : smart_ptr<Structure>*
- *clone_type (type: smart_ptr<TypeDecl>) : smart_ptr<TypeDecl>*
- *clone_variable (variable: smart_ptr<Variable>) : smart_ptr<Variable>*

ast::**clone_expression(expression: smart_ptr<Expression>) : smart_ptr<Expression>**()

Clones an Expression along with all its subexpressions and corresponding type information.

Arguments

- **expression** : smart_ptr< *Expression*> implicit

ast::**clone_file_info(name: string; tab_size: int) : FileInfo?**()

Clones a FileInfo structure.

Arguments

- **name** : string implicit
- **tab_size** : int

clone_function

`ast::`**`clone_function(function: smart_ptr<Function>) : smart_ptr<Function>`**`()`

Clones a Function and all of its contents.

Arguments

- **function** : smart_ptr< *Function*> implicit

`ast::`**`clone_function(fn: Function?) : FunctionPtr`**`()`

`ast::`**`clone_structure(structure: Structure const?) : smart_ptr<Structure>`**`()`

Returns a deep clone of the specified Structure.

Arguments

- **structure** : *Structure*? implicit

`ast::`**`clone_type(type: smart_ptr<TypeDecl>) : smart_ptr<TypeDecl>`**`()`

Clones a TypeDecl along with all its subtypes.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit

`ast::`**`clone_variable(variable: smart_ptr<Variable>) : smart_ptr<Variable>`**`()`

Clones a Variable and all of its contents.

Arguments

- **variable** : smart_ptr< *Variable*> implicit

10.2.19 Mangled name

- *get_mangled_name (function: smart_ptr<Function>) : string*
- *get_mangled_name (type: smart_ptr<TypeDecl>) : string*
- *get_mangled_name (variable: smart_ptr<Variable>) : string*
- *get_mangled_name (variable: smart_ptr<ExprBlock>) : string*
- *get_mangled_name (decl: TypeDecl?) : auto*
- *get_mangled_name (fn: Function?) : auto*
- *get_mangled_name (decl: Variable?) : auto*
- *get_mangled_name (decl: ExprBlock?) : auto*
- *parse_mangled_name (txt: string; lib: ModuleGroup; thisModule: Module?) : smart_ptr<TypeDecl>*

get_mangled_name

ast::**get_mangled_name(function: smart_ptr<Function>) : string**()

Returns the mangled name of the specified object.

Arguments

- **function** : smart_ptr< *Function*> implicit

ast::**get_mangled_name(type: smart_ptr<TypeDecl>) : string**()

ast::**get_mangled_name(variable: smart_ptr<Variable>) : string**()

ast::**get_mangled_name(variable: smart_ptr<ExprBlock>) : string**()

ast::**get_mangled_name(decl: TypeDecl?) : auto**()

ast::**get_mangled_name(fn: Function?) : auto**()

ast::**get_mangled_name(decl: Variable?) : auto**()

ast::**get_mangled_name(decl: ExprBlock?) : auto**()

---

ast::**parse_mangled_name(txt: string; lib: ModuleGroup; thisModule: Module?) : smart_ptr<TypeDecl>**()

Parses a mangled name string and creates the corresponding TypeDecl.

Arguments

- **txt** : string implicit
- **lib** : *ModuleGroup* implicit
- **thisModule** : *Module*? implicit

10.2.20 Size and offset

- *any_array_size (array: void?) : int*
- *any_table_size (table: void?) : int*
- *get_handled_type_field_offset (type: smart_ptr<TypeAnnotation>; field: string) : uint*
- *get_tuple_field_offset (typle: smart_ptr<TypeDecl>; index: int) : int*
- *get_variant_field_offset (variant: smart_ptr<TypeDecl>; index: int) : int*

ast::**any_array_size(array: void?) : int**()

Returns the size of an array from a pointer to an *array*<> object.

Arguments

- **array** : void? implicit

ast::**any_table_size(table: void?) : int**()

Returns the size of a table from a pointer to a *table*<> object.

Arguments

- **table** : void? implicit

ast::**get_handled_type_field_offset(type: smart_ptr<TypeAnnotation>; field: string) : uint()**

Returns the byte offset of a field in a ManagedStructure handled type.

Arguments

- **type** : smart_ptr< *TypeAnnotation*> implicit
- **field** : string implicit

ast::**get_tuple_field_offset(typle: smart_ptr<TypeDecl>; index: int) : int()**

Returns the byte offset of a tuple field.

Arguments

- **typle** : smart_ptr< *TypeDecl*> implicit
- **index** : int

ast::**get_variant_field_offset(variant: smart_ptr<TypeDecl>; index: int) : int()**

Returns the byte offset of a variant field.

Arguments

- **variant** : smart_ptr< *TypeDecl*> implicit
- **index** : int

10.2.21 Evaluations

- *eval_single_expression (expr: smart_ptr<Expression> const&; ok: bool&) : float4*

ast::**eval_single_expression(expr: smart_ptr<Expression> const&; ok: bool&) : float4()**

Warning: This is unsafe operation.

Simulates and evaluates a single expression on a separate context.

Arguments

- **expr** : smart_ptr< *Expression*>& implicit
- **ok** : bool& implicit

10.2.22 Error reporting

- *macro_error (porogram: smart_ptr<Program>; at: LineInfo; message: string)*

ast::**macro_error**(*porogram: smart_ptr<Program>; at: LineInfo; message: string*)

Reports an error to the currently compiling program during the active compilation pass.

Arguments

- **porogram** : smart_ptr< *Program*> implicit
- **at** : *LineInfo* implicit
- **message** : string implicit

10.2.23 Location and context

- *collect_dependencies (function: smart_ptr<Function>; block: block<(array<Function?>;array<Variable?>):void>)*
- *force_at (function: smart_ptr<Function> const&; at: LineInfo)*
- *force_at (expression: smart_ptr<Expression> const&; at: LineInfo)*
- *get_ast_context (program: smart_ptr<Program>; expression: smart_ptr<Expression>; block: block<(bool;AstContext):void>)*

`ast::`**`collect_dependencies`**(*function: smart_ptr<Function>; block: block<(array<Function?>;array<Variable?>):void>*)

Collects dependencies of a given function, including other functions it calls and global variables it accesses.

Arguments

- **function** : smart_ptr< *Function*> implicit
- **block** : block<(array< *Function*?>;array< *Variable*?>):void> implicit

force_at

`ast::`**`force_at`**(*function: smart_ptr<Function> const&; at: LineInfo*)

Replaces line info in an expression, its subexpressions, and their types.

Arguments

- **function** : smart_ptr< *Function*>& implicit
- **at** : *LineInfo* implicit

`ast::`**`force_at`**(*expression: smart_ptr<Expression> const&; at: LineInfo*)

`ast::`**`get_ast_context`**(*program: smart_ptr<Program>; expression: smart_ptr<Expression>; block: block<(bool;AstContext):void>*)

Returns the AstContext for a given expression, including the current function, loops, blocks, scopes, and with sections.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **expression** : smart_ptr< *Expression*> implicit
- **block** : block<(bool; *AstContext*):void> implicit

10.2.24 Use queries

- *get_use_functions (func: smart_ptr<Function>; block: block<(smart_ptr<Function>):void>)*
- *get_use_global_variables (func: smart_ptr<Function>; block: block<(smart_ptr<Variable>):void>)*

ast::**get_use_functions**(*func: smart_ptr<Function>; block: block<(smart_ptr<Function>):void>*)

Invokes a block with the list of all functions called by the specified function.

Arguments

- **func** : smart_ptr< *Function*> implicit
- **block** : block<(smart_ptr< *Function*>):void> implicit

ast::**get_use_global_variables**(*func: smart_ptr<Function>; block: block<(smart_ptr<Variable>):void>*)

Invokes a block with the list of all global variables accessed by the specified function.

Arguments

- **func** : smart_ptr< *Function*> implicit
- **block** : block<(smart_ptr< *Variable*>):void> implicit

10.2.25 Log

- *to_compilation_log (text: string)*

ast::**to_compilation_log**(*text: string*)

Writes a message to the compilation log from a macro during compilation.

Arguments

- **text** : string implicit

10.2.26 Removal

- *remove_structure (module: Module?; structure: smart_ptr<Structure>&) : bool*

ast::**remove_structure(module: Module?; structure: smart_ptr<Structure>&) : bool**()

Removes a structure declaration from the specified module.

Arguments

- **module** : *Module*? implicit
- **structure** : smart_ptr< *Structure*>& implicit

10.2.27 Properties

- *can_access_global_variable (variable: smart_ptr<Variable> const&; module: Module?; thisModule: Module?) : bool*
- *get_aot_arg_prefix (func: Function?; call: ExprCallFunc?; argIndex: int) : string*
- *get_aot_arg_suffix (func: Function?; call: ExprCallFunc?; argIndex: int) : string*
- *get_aot_name (func: Function?; call: ExprCallFunc?) : string*
- *get_current_search_module (program: Program?; function: Function?; moduleName: string) : Module?*
- *get_field_type (type: smart_ptr<TypeDecl>; fieldName: string; constant: bool) : smart_ptr<TypeDecl>*
- *get_func_aot_prefix (ann: FunctionAnnotation?; stg: StringBuilderWriter?; call: ExprCallFunc?)*

- *get_function_aot_hash (fun: Function const?) : uint64*
- *get_function_hash_by_id (fun: Function?; id: int; pctx: void?) : uint64*
- *get_handled_type_field_type (type: smart_ptr<TypeAnnotation>; field: string) : TypeInfo?*
- *get_handled_type_field_type_declaration (type: smart_ptr<TypeAnnotation>; field: string; isConst: bool) : smart_ptr<TypeDecl>*
- *get_handled_type_index_type_declaration (type: TypeAnnotation?; src: Expression?; idx: Expression?) : smart_ptr<TypeDecl>*
- *get_struct_aot_prefix (ann: StructureAnnotation?; structure: Structure?; args: AnnotationArgumentList; stg: StringBuilderWriter?)*
- *get_structure_alias (structure: Structure?; aliasName: string) : smart_ptr<TypeDecl>*
- *get_underlying_value_type (type: smart_ptr<TypeDecl>) : smart_ptr<TypeDecl>*
- *get_vector_length (vec: void?; type: smart_ptr<TypeDecl>) : int*
- *get_vector_ptr_at_index (vec: void?; type: TypeDecl?; idx: int) : void?*
- *has_field (type: smart_ptr<TypeDecl>; fieldName: string; constant: bool) : bool*
- *is_expr_const (expression: smart_ptr<Expression> const&) : bool*
- *is_expr_like_call (expression: smart_ptr<Expression> const&) : bool*
- *is_same_type (leftType: smart_ptr<TypeDecl>; rightType: smart_ptr<TypeDecl>; refMatters: RefMatters; constMatters: ConstMatters; tempMatters: TemporaryMatters) : bool*
- *is_same_type (argType: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>; refMatters: bool; constMatters: bool; temporaryMatters: bool; allowSubstitute: bool) : bool*
- *is_temp_type (type: smart_ptr<TypeDecl>; refMatters: bool) : bool*
- *is_visible_directly (from_module: Module?; which_module: Module?) : bool*

`ast::can_access_global_variable(variable: smart_ptr<Variable> const&; module: Module?; thisModule: Modul`

Returns true if a global variable is accessible from the specified module.

Arguments

- **variable** : smart_ptr< *Variable*>& implicit
- **module** : *Module*? implicit
- **thisModule** : *Module*? implicit

`ast::get_aot_arg_prefix(func: Function?; call: ExprCallFunc?; argIndex: int) : string()`

Returns the AOT argument prefix string for the specified function.

Arguments

- **func** : *Function*? implicit
- **call** : *ExprCallFunc*? implicit
- **argIndex** : int

`ast::get_aot_arg_suffix(func: Function?; call: ExprCallFunc?; argIndex: int) : string()`

Returns the AOT argument suffix string for the specified function.

Arguments

- **func** : *Function*? implicit
- **call** : *ExprCallFunc*? implicit
- **argIndex** : int

ast::**get_aot_name(func: Function?; call: ExprCallFunc?) : string**()

Returns the AOT-generated name for the specified function.

Arguments

- **func** : *Function*? implicit
- **call** : *ExprCallFunc*? implicit

ast::**get_current_search_module(program: Program?; function: Function?; moduleName: string) : Module?**()

Returns the module currently being searched for a function by name, correctly resolving special names like "", "_", "*", and "__".

Arguments

- **program** : *Program*? implicit
- **function** : *Function*? implicit
- **moduleName** : string implicit

ast::**get_field_type(type: smart_ptr<TypeDecl>; fieldName: string; constant: bool) : smart_ptr<TypeDecl>**

Returns the type of a field if the target is a structure, variant, tuple, handled type, or pointer to any of those, or null otherwise.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **fieldName** : string implicit
- **constant** : bool

ast::**get_func_aot_prefix**(*ann: FunctionAnnotation?; stg: StringBuilderWriter?; call: ExprCallFunc?*)

Returns the AOT function prefix string for the specified function.

Arguments

- **ann** : *FunctionAnnotation*? implicit
- **stg** : *StringBuilderWriter*? implicit
- **call** : *ExprCallFunc*? implicit

ast::**get_function_aot_hash(fun: Function const?) : uint64**()

Returns the hash of a function used for AOT matching.

Arguments

- **fun** : *Function*? implicit

ast::**get_function_hash_by_id(fun: Function?; id: int; pctx: void?) : uint64**()

Returns the hash of a function given its unique identifier.

Arguments

- **fun** : *Function*? implicit

- **id** : int
- **pctx** : void? implicit

ast::**get_handled_type_field_type(type: smart_ptr<TypeAnnotation>; field: string) : TypeInfo?**()

Returns the type of a field in a ManagedStructure handled type.

Arguments

- **type** : smart_ptr< *TypeAnnotation*> implicit
- **field** : string implicit

ast::**get_handled_type_field_type_declaration(type: smart_ptr<TypeAnnotation>; field: string; isConst: bo**

Returns the type declaration of a field in a ManagedStructure handled type.

Arguments

- **type** : smart_ptr< *TypeAnnotation*> implicit
- **field** : string implicit
- **isConst** : bool

ast::**get_handled_type_index_type_declaration(type: TypeAnnotation?; src: Expression?; idx: Expression?)**

Returns the type declaration of the index operator for a handled type.

Arguments

- **type** : *TypeAnnotation*? implicit
- **src** : *Expression*? implicit
- **idx** : *Expression*? implicit

ast::**get_struct_aot_prefix**(*ann: StructureAnnotation?; structure: Structure?; args: AnnotationArgumentList; stg: StringBuilderWriter?*)

Returns the AOT prefix string for the specified structure.

Arguments

- **ann** : *StructureAnnotation*? implicit
- **structure** : *Structure*? implicit
- **args** : *AnnotationArgumentList* implicit
- **stg** : *StringBuilderWriter*? implicit

ast::**get_structure_alias(structure: Structure?; aliasName: string) : smart_ptr<TypeDecl>**()

Finds and returns a structure alias type by its alias name.

Arguments

- **structure** : *Structure*? implicit
- **aliasName** : string implicit

ast::**get_underlying_value_type(type: smart_ptr<TypeDecl>) : smart_ptr<TypeDecl>**()

Returns the daslang type aliased by a ManagedValue handled type.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit

ast::**get_vector_length(vec: void?; type: smart_ptr<TypeDecl>) : int**()

Returns the length of a vector given a pointer to the vector object and its TypeDeclPtr.

Arguments

- **vec** : void? implicit
- **type** : smart_ptr< *TypeDecl*> implicit

ast::**get_vector_ptr_at_index(vec: void?; type: TypeDecl?; idx: int) : void?**()

Returns a pointer to the vector element at the specified index given a pointer to the vector object and its TypeDeclPtr.

Arguments

- **vec** : void? implicit
- **type** : *TypeDecl*? implicit
- **idx** : int

ast::**has_field(type: smart_ptr<TypeDecl>; fieldName: string; constant: bool) : bool**()

Returns true if a structure, variant, tuple, handled type, or pointer to any of those has the specified field.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **fieldName** : string implicit
- **constant** : bool

ast::**is_expr_const(expression: smart_ptr<Expression> const&) : bool**()

Returns true if the expression is or inherits from ExprConst.

Arguments

- **expression** : smart_ptr< *Expression*>& implicit

ast::**is_expr_like_call(expression: smart_ptr<Expression> const&) : bool**()

Returns true if the expression is or inherits from ExprLooksLikeCall.

Arguments

- **expression** : smart_ptr< *Expression*>& implicit

is_same_type

ast::**is_same_type(leftType: smart_ptr<TypeDecl>; rightType: smart_ptr<TypeDecl>; refMatters: RefMatters**

Compares two types using the given comparison parameters and returns true if they match.

Arguments

- **leftType** : smart_ptr< *TypeDecl*> implicit
- **rightType** : smart_ptr< *TypeDecl*> implicit
- **refMatters** : *RefMatters*
- **constMatters** : *ConstMatters*

- **tempMatters** : *TemporaryMatters*

ast::**is_same_type(argType: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>; refMatters: bool; constMa**

---

ast::**is_temp_type(type: smart_ptr<TypeDecl>; refMatters: bool) : bool**()

Returns true if the specified type can be temporary.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **refMatters** : bool

ast::**is_visible_directly(from_module: Module?; which_module: Module?) : bool**()

Returns true if one module is directly visible from another module.

Arguments

- **from_module** : *Module*? implicit
- **which_module** : *Module*? implicit

10.2.28 Infer

- *infer_generic_type (type: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>; topLevel: bool; isPassType: bool) : smart_ptr<TypeDecl>*
- *update_alias_map (program: smart_ptr<Program>; argType: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>)*

ast::**infer_generic_type(type: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>; topLevel: bool; isPass**

Infers a concrete type from a generic type declaration and a pass type.

Arguments

- **type** : smart_ptr< *TypeDecl*> implicit
- **passType** : smart_ptr< *TypeDecl*> implicit
- **topLevel** : bool
- **isPassType** : bool

ast::**update_alias_map**(*program: smart_ptr<Program>; argType: smart_ptr<TypeDecl>; passType: smart_ptr<TypeDecl>*)

Updates the alias map for the specified type during inference.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **argType** : smart_ptr< *TypeDecl*> implicit
- **passType** : smart_ptr< *TypeDecl*> implicit

10.2.29 Module queries

- *module_find_annotation (module: Module const?; name: string) : smart_ptr<Annotation>*
- *module_find_structure (program: Module const?; name: string) : Structure?*
- *module_find_type_annotation (module: Module const?; name: string) : TypeAnnotation?*
- *not_inferred (function: Function?)*

ast::**module_find_annotation(module: Module const?; name: string) : smart_ptr<Annotation>**()

Finds an annotation by name in the specified module.

Arguments

- **module** : *Module*? implicit
- **name** : string implicit

ast::**module_find_structure(program: Module const?; name: string) : Structure?**()

Finds a structure by name in the specified module.

Arguments

- **program** : *Module*? implicit
- **name** : string implicit

ast::**module_find_type_annotation(module: Module const?; name: string) : TypeAnnotation?**()

Finds a type annotation by name in the specified module.

Arguments

- **module** : *Module*? implicit
- **name** : string implicit

ast::**not_inferred**(*function: Function?*)

Marks a function as modified by a macro so that it will be inferred again.

Arguments

- **function** : *Function*? implicit

10.2.30 Debug info helpers

- *debug_helper_find_struct_cppname (helper: smart_ptr<DebugInfoHelper> const&; struct_info: StructInfo?) : string*
- *debug_helper_find_type_cppname (helper: smart_ptr<DebugInfoHelper> const&; type_info: TypeInfo?) : string*
- *debug_helper_iter_enums (helper: smart_ptr<DebugInfoHelper>; blk: block<(string;EnumInfo?):void>)*
- *debug_helper_iter_funcs (helper: smart_ptr<DebugInfoHelper>; blk: block<(string;FuncInfo?):void>)*
- *debug_helper_iter_structs (helper: smart_ptr<DebugInfoHelper>; blk: block<(string;StructInfo?):void>)*
- *debug_helper_iter_types (helper: smart_ptr<DebugInfoHelper>; blk: block<(string;TypeInfo?):void>)*
- *debug_helper_iter_vars (helper: smart_ptr<DebugInfoHelper>; blk: block<(string;VarInfo?):void>)*

ast::**debug_helper_find_struct_cppname(helper: smart_ptr<DebugInfoHelper> const&; struct_info: StructInf**

Finds a structure in the DebugInfoHelper and returns its C++ name.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*>& implicit
- **struct_info** : *StructInfo*? implicit

ast::**debug_helper_find_type_cppname(helper: smart_ptr<DebugInfoHelper> const&; type_info: TypeInfo?) : s**

Finds a type in the DebugInfoHelper and returns its C++ name.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*>& implicit
- **type_info** : *TypeInfo*? implicit

ast::**debug_helper_iter_enums**(*helper: smart_ptr<DebugInfoHelper>; blk: block<(string;EnumInfo?):void>*)

Iterates through all enumerations in the DebugInfoHelper, invoking the provided block for each one.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **blk** : block<(string; *EnumInfo*?):void> implicit

ast::**debug_helper_iter_funcs**(*helper: smart_ptr<DebugInfoHelper>; blk: block<(string;FuncInfo?):void>*)

Iterates through all functions in the DebugInfoHelper, invoking the provided block for each one.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **blk** : block<(string; *FuncInfo*?):void> implicit

ast::**debug_helper_iter_structs**(*helper: smart_ptr<DebugInfoHelper>; blk: block<(string;StructInfo?):void>*)

Iterates through all structures in the DebugInfoHelper, invoking the provided block for each one.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **blk** : block<(string; *StructInfo*?):void> implicit

ast::**debug_helper_iter_types**(*helper: smart_ptr<DebugInfoHelper>; blk: block<(string;TypeInfo?):void>*)

Iterates through all types in the DebugInfoHelper, invoking the provided block for each one.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit
- **blk** : block<(string; *TypeInfo*?):void> implicit

ast::**debug_helper_iter_vars**(*helper: smart_ptr<DebugInfoHelper>; blk: block<(string;VarInfo?):void>*)

Iterates through all variables in the DebugInfoHelper, invoking the provided block for each one.

Arguments

- **helper** : smart_ptr< *DebugInfoHelper*> implicit

- **blk** : block<(string; *VarInfo*?):void> implicit

10.2.31 AOT support

- *aot_need_type_info (macro: TypeInfoMacro const?; expr: smart_ptr<Expression>) : bool*
- *aot_previsit_get_field (ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string)*
- *aot_previsit_get_field_ptr (ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string)*
- *aot_require (mod: Module?; ss: StringBuilderWriter?) : bool*
- *aot_type_ann_get_field_ptr (ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string)*
- *aot_visit_get_field (ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string)*
- *getInitSemanticHashWithDep (program: smart_ptr<Program>; init: uint64) : uint64*
- *macro_aot_infix (macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>) : bool*
- *write_aot_body (structure: StructureAnnotation?; st: smart_ptr<Structure>; args: AnnotationArgumentList; writer: StringBuilderWriter?)*
- *write_aot_macro_prefix (macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>)*
- *write_aot_macro_suffix (macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>)*
- *write_aot_suffix (structure: StructureAnnotation?; st: smart_ptr<Structure>; args: AnnotationArgumentList; writer: StringBuilderWriter?)*

`ast::`**`aot_need_type_info(macro: TypeInfoMacro const?; expr: smart_ptr<Expression>) : bool`**`()`

Returns true if a *TypeInfo?* is needed for the specified type in a typeinfo expression.

Arguments

- **macro** : *TypeInfoMacro*? implicit
- **expr** : smart_ptr< *Expression*> implicit

`ast::`**`aot_previsit_get_field`**`(`*ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string*`)`

Performs the pre-visit step for field access during AOT code generation.

Arguments

- **ann** : *TypeAnnotation*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **name** : string implicit

`ast::`**`aot_previsit_get_field_ptr`**`(`*ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string*`)`

Performs the pre-visit step for field pointer access during AOT code generation.

Arguments

- **ann** : *TypeAnnotation*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **name** : string implicit

ast::**aot_require(mod: Module?; ss: StringBuilderWriter?) : bool**()

Writes data to the require section of an AOT module.

Arguments

- **mod** : *Module*? implicit
- **ss** : *StringBuilderWriter*? implicit

ast::**aot_type_ann_get_field_ptr**(*ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string*)

Returns the access symbol string for a field, such as -> for pointer types or . for value types.

Arguments

- **ann** : *TypeAnnotation*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **name** : string implicit

ast::**aot_visit_get_field**(*ann: TypeAnnotation?; ss: StringBuilderWriter?; name: string*)

Performs the visit step for field access during AOT code generation.

Arguments

- **ann** : *TypeAnnotation*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **name** : string implicit

ast::**getInitSemanticHashWithDep(program: smart_ptr<Program>; init: uint64) : uint64**()

Returns the initialization semantic hash including dependencies for the entire program.

Arguments

- **program** : smart_ptr< *Program*> implicit
- **init** : uint64

ast::**macro_aot_infix(macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>) : boo**

Returns true if the macro requires an AOT infix operator for the specified handled type.

Arguments

- **macro** : *TypeInfoMacro*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **expr** : smart_ptr< *Expression*> implicit

ast::**write_aot_body**(*structure: StructureAnnotation?; st: smart_ptr<Structure>; args: AnnotationArgumentList; writer: StringBuilderWriter?*)

Writes the AOT body code for the specified StructureAnnotation.

Arguments

- **structure** : *StructureAnnotation*? implicit
- **st** : smart_ptr< *Structure*> implicit
- **args** : *AnnotationArgumentList* implicit

- **writer** : *StringBuilderWriter*? implicit

ast::**write_aot_macro_prefix**(*macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>*)

Writes the AOT macro prefix code for the specified TypeInfoMacro.

Arguments

- **macro** : *TypeInfoMacro*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **expr** : smart_ptr< *Expression*> implicit

ast::**write_aot_macro_suffix**(*macro: TypeInfoMacro?; ss: StringBuilderWriter?; expr: smart_ptr<Expression>*)

Writes the AOT macro suffix code for the specified TypeInfoMacro.

Arguments

- **macro** : *TypeInfoMacro*? implicit
- **ss** : *StringBuilderWriter*? implicit
- **expr** : smart_ptr< *Expression*> implicit

ast::**write_aot_suffix**(*structure: StructureAnnotation?; st: smart_ptr<Structure>; args: AnnotationArgumentList; writer: StringBuilderWriter?*)

Writes the AOT suffix code for the specified StructureAnnotation.

Arguments

- **structure** : *StructureAnnotation*? implicit
- **st** : smart_ptr< *Structure*> implicit
- **args** : *AnnotationArgumentList* implicit
- **writer** : *StringBuilderWriter*? implicit

10.2.32 String builder writer

- *string_builder_clear (ss: StringBuilderWriter?)*
- *string_builder_str (ss: StringBuilderWriter?) : string*

ast::**string_builder_clear**(*ss: StringBuilderWriter?*)

Clears a StringBuilder object given a pointer to it.

Arguments

- **ss** : *StringBuilderWriter*? implicit

ast::**string_builder_str(ss: StringBuilderWriter?) : string**()

Returns the accumulated string from a StringBuilder object given a pointer to it.

Arguments

- **ss** : *StringBuilderWriter*? implicit

10.3 Boost package for the AST

The AST_BOOST module provides high-level utilities for working with the AST. It includes helpers for creating expressions, types, and declarations, quote-based AST construction, and common AST query and transformation patterns used by macro authors.

All functions and symbols are in "ast_boost" module, use require to get access to it.

```
require daslib/ast_boost
```

10.3.1 Type aliases

ast_boost::**AnnotationDeclarationPtr = smart_ptr<AnnotationDeclaration>**

Type alias for `smart_ptr<AnnotationDeclaration>`, used when constructing or attaching annotation declarations to functions, blocks, or structures.

ast_boost::**bitfield DebugExpressionFlags**

Fields

- **refCount** (0x1) - Bitfield controlling `debug_expression` output — currently has a single `refCount` flag that includes smart pointer reference counts in the dump.

10.3.2 Function annotations

ast_boost::**macro**

The `[macro]` function annotation — marks a function to run only during macro module compilation, gating its body behind `is_compiling_macros`.

ast_boost::**tag_function**

The `[tag_function]` function annotation — attaches named tags to a function so that `[tag_function_macro]`-based annotations can discover and process it.

10.3.3 Variant macros

ast_boost::**better_rtti_in_expr**

Variant macro that enables improved RTTI type matching in *is* and *as* expressions. Varian macro better_rtti_in_expr

10.3.4 Structure macros

ast_boost::**function_macro**

The `[function_macro]` structure annotation — registers an `AstFunctionAnnotation` subclass as a named function annotation available to the compiler.

ast_boost::**block_macro**

The `[block_macro]` structure annotation — registers an `AstBlockAnnotation` subclass as a named block-level annotation available to the compiler.

ast_boost::**structure_macro**

The [structure_macro] structure annotation — registers an AstStructureAnnotation subclass as a named annotation applicable to structures and classes.

ast_boost::**enumeration_macro**

The [enumeration_macro] structure annotation — registers an AstEnumerationAnnotation subclass as a named annotation applicable to enumerations.

ast_boost::**contract**

The [contract] structure annotation — registers an AstFunctionAnnotation subclass as a named function contract that validates arguments or return values.

ast_boost::**reader_macro**

The [reader_macro] structure annotation — registers an AstReaderMacro subclass as a named reader macro invoked by the %name~...~~ syntax during parsing.

ast_boost::**comment_reader**

The [comment_reader] structure annotation — registers an AstCommentReader subclass as a named comment reader invoked during parsing.

ast_boost::**call_macro**

The [call_macro] structure annotation — registers an AstCallMacro subclass as a named call-expression macro available during compilation.

ast_boost::**typeinfo_macro**

The [typeinfo_macro] structure annotation — registers an AstTypeInfoMacro subclass as a named macro that extends the typeinfo(name ...) built-in.

ast_boost::**variant_macro**

The [variant_macro] structure annotation — registers an AstVariantMacro subclass as a named macro that can customize is, as, and ?as variant operations.

ast_boost::**for_loop_macro**

The [for_loop_macro] structure annotation — registers an AstForLoopMacro subclass as a named macro that can transform for loop expressions.

ast_boost::**capture_macro**

The [capture_macro] structure annotation — registers an AstCaptureMacro subclass as a named capture macro that can customize lambda capture behavior.

ast_boost::**type_macro**

The [type_macro] structure annotation — registers an AstTypeMacro subclass as a named macro that can intercept and transform type expressions.

ast_boost::**simulate_macro**

The [simulate_macro] structure annotation — registers an AstSimulateMacro subclass as a named macro invoked during the simulation (code generation) phase.

ast_boost::**tag_structure**

The [tag_structure] structure annotation — attaches named boolean tags to a structure, allowing macro code to discover and process tagged structures.

ast_boost::**tag_function_macro**

The [tag_function_macro] structure annotation — registers an AstFunctionAnnotation that is automatically applied to every function carrying a matching [tag_function(tag)] tag.

ast_boost::**infer_macro**

The [infer_macro] structure annotation — registers an AstPassMacro subclass that is invoked during the type inference compilation pass.

ast_boost::**dirty_infer_macro**

The [dirty_infer_macro] structure annotation — registers an AstPassMacro subclass that is invoked during the dirty infer compilation pass.

ast_boost::**optimization_macro**

The [optimization_macro] structure annotation — registers an AstPassMacro subclass that is invoked during the optimization compilation pass.

ast_boost::**lint_macro**

The [lint_macro] structure annotation — registers an AstPassMacro subclass that is invoked during the lint compilation pass.

ast_boost::**global_lint_macro**

The [global_lint_macro] structure annotation — registers an AstPassMacro subclass that is invoked during the global lint compilation pass.

10.3.5 Classes

ast_boost::**MacroMacro : AstFunctionAnnotation**

Implements the [macro] function annotation, which wraps the function body so it only executes during macro module compilation.

MacroMacro.**apply(func: FunctionPtr; group: ModuleGroup; args: AnnotationArgumentList; errors: das_strin**

Wraps the annotated function body in an is_compiling_macros guard and sets macroInit flag so it only runs during macro module compilation.

Arguments

- **func** : *FunctionPtr*
- **group** : *ModuleGroup*
- **args** : *AnnotationArgumentList*
- **errors** : *das_string*

ast_boost::**TagFunctionAnnotation : AstFunctionAnnotation**

Implements the [tag_function] function annotation, which attaches named boolean tags to functions so they can be discovered and processed by [tag_function_macro].

TagFunctionAnnotation.**apply(func: FunctionPtr; group: ModuleGroup; args: AnnotationArgumentList; errors**

Validates that all [tag_function(...)] annotation arguments are tag names (boolean flags) and rejects any non-boolean arguments with an error.

Arguments

- **func** : *FunctionPtr*
- **group** : *ModuleGroup*
- **args** : *AnnotationArgumentList*
- **errors** : *das_string*

ast_boost::**TagStructureAnnotation : AstStructureAnnotation**

Implements the [tag_structure] structure annotation, which attaches named boolean tags to structures for later discovery by macro code.

TagStructureAnnotation.**apply(str: StructurePtr; group: ModuleGroup; args: AnnotationArgumentList; error**

Validates that all [tag_structure(...)] annotation arguments are tag names (boolean flags) and rejects any non-boolean arguments with an error.

Arguments

- **str** : *StructurePtr*
- **group** : *ModuleGroup*
- **args** : *AnnotationArgumentList*
- **errors** : *das_string*

ast_boost::**SetupAnyAnnotation : AstStructureAnnotation**

This is base class for any annotation or macro setup.

Fields

- **annotation_function_call** : string = "" - Function call name, which is used to setup any annotation.
- **name** : string - Name of the annotation to setup.

SetupAnyAnnotation.**apply(st: StructurePtr; group: ModuleGroup; args: AnnotationArgumentList; errors: da**

Generates a macro-init function that constructs an instance of the annotated class and registers it with the compiler under the specified name.

Arguments

- **st** : *StructurePtr*
- **group** : *ModuleGroup*
- **args** : *AnnotationArgumentList*
- **errors** : *das_string*

SetupAnyAnnotation.**setup_call**(*st: StructurePtr; cll: smart_ptr<ExprCall>*)

Populates the registration call arguments — by default adds the annotation name as a string constant; overridden in subclasses to add extra parameters.

Arguments

- **st** : *StructurePtr*
- **cll** : smart_ptr< *ExprCall*>

ast_boost::**SetupFunctionAnnotation : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_function_annotation" - Base class for creating function annotations via the `[function_macro]` structure annotation; registers an `AstFunctionAnnotation` with the compiler.

ast_boost::**SetupBlockAnnotation : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_block_annotation" - Base class for creating block annotations via the `[block_macro]` structure annotation; registers an `AstBlockAnnotation` with the compiler.

ast_boost::**SetupStructureAnnotation : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_structure_annotation" - Base class for creating structure annotations via the `[structure_macro]` structure annotation; registers an `AstStructureAnnotation` with the compiler.

ast_boost::**SetupEnumerationAnnotation : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_enumeration_annotation" - Base class for creating enumeration annotations via the `[enumeration_macro]` structure annotation; registers an `AstEnumerationAnnotation` with the compiler.

ast_boost::**SetupContractAnnotation : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_contract_annotation" - Base class for creating function contract annotations via the `[contract]` structure annotation; registers an `AstFunctionAnnotation` as a contract.

ast_boost::**SetupReaderMacro : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_reader_macro" - Base class for creating reader macros via the `[reader_macro]` structure annotation; registers an `AstReaderMacro` with the compiler.

ast_boost::**SetupCommentReader : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_comment_reader" - Base class for creating comment readers via the `[comment_reader]` structure annotation; registers an `AstCommentReader` with the compiler.

ast_boost::**SetupVariantMacro : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_variant_macro" - Base class for creating variant macros via the `[variant_macro]` structure annotation; registers an `AstVariantMacro` with the compiler.

`ast_boost::`**`SetupForLoopMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_for_loop_macro" - Base class for creating for-loop macros via the `[for_loop_macro]` structure annotation; registers an `AstForLoopMacro` with the compiler.

`ast_boost::`**`SetupCaptureMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_capture_macro" - Base class for creating capture macros via the `[capture_macro]` structure annotation; registers an `AstCaptureMacro` with the compiler.

`ast_boost::`**`SetupTypeMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_type_macro" - Base class for creating type macros via the `[type_macro]` structure annotation; registers an `AstTypeMacro` with the compiler.

`ast_boost::`**`SetupSimulateMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_simulate_macro" - Base class for creating simulate macros via the `[simulate_macro]` structure annotation; registers an `AstSimulateMacro` with the compiler.

`ast_boost::`**`SetupCallMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_call_macro" - Base class for creating call macros via the `[call_macro]` structure annotation; registers an `AstCallMacro` with the compiler.

`ast_boost::`**`SetupTypeInfoMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_typeinfo_macro" - Base class for creating typeinfo macros via the `[typeinfo_macro]` structure annotation; registers an `AstTypeInfoMacro` with the compiler.

`ast_boost::`**`SetupInferMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_infer_macro" - Base class for creating infer pass macros via the `[infer_macro]` structure annotation; registers an `AstPassMacro` that runs during the type inference pass.

`ast_boost::`**`SetupDirtyInferMacro : SetupAnyAnnotation`**

Fields

- **annotation_function_call** : string = "add_new_dirty_infer_macro" - Base class for creating dirty-infer pass macros via the `[dirty_infer_macro]` structure annotation; registers an `AstPassMacro` that runs during the dirty infer pass.

ast_boost::**SetupLintMacro : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_lint_macro" - Base class for creating lint pass macros via the `[lint_macro]` structure annotation; registers an `AstPassMacro` that runs during the lint pass.

ast_boost::**SetupGlobalLintMacro : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_global_lint_macro" - Base class for creating global lint pass macros via the `[global_lint_macro]` structure annotation; registers an `AstPassMacro` that runs during the global lint pass.

ast_boost::**SetupOptimizationMacro : SetupAnyAnnotation**

Fields

- **annotation_function_call** : string = "add_new_optimization_macro" - Base class for creating optimization pass macros via the `[optimization_macro]` structure annotation; registers an `AstPassMacro` that runs during the optimization pass.

ast_boost::**TagFunctionMacro : SetupAnyAnnotation**

[tag_function_macro] implementation.

Fields

- **annotation_function_call** : string = "setup_tag_annotation" - Name of the function call, which setups up the annotation.
- **tag** : string - Name of the tag.

TagFunctionMacro.**apply(st: StructurePtr; group: ModuleGroup; args: AnnotationArgumentList; errors: das_s**

Extends `SetupAnyAnnotation` apply to extract the required `tag` argument and register a `setup_tag_annotation` call that links the annotation to tagged functions.

Arguments

- **st** : *StructurePtr*
- **group** : *ModuleGroup*
- **args** : *AnnotationArgumentList*
- **errors** : *das_string*

TagFunctionMacro.**setup_call**(*st: StructurePtr; cll: smart_ptr<ExprCall>*)

Overrides the default `setup_call` to pass both the annotation `name` and the `tag` string as arguments to `setup_tag_annotation`.

Arguments

- **st** : *StructurePtr*
- **cll** : smart_ptr< *ExprCall*>

ast_boost::**BetterRttiVisitor : AstVariantMacro**

An AstVariantMacro that replaces is, as, and ?as variant operations on Expression subclasses with runtime __rtti string checks and casts.

BetterRttiVisitor.**visitExprIsVariant(prog: ProgramPtr; mod: Module?; expr: smart_ptr<ExprIsVariant>) :**

Visitor override that replaces `expr is Type` on Expression subclasses with an __rtti string comparison, returning true if the runtime type matches.

Arguments

- **prog** : *ProgramPtr*
- **mod** : *Module*?
- **expr** : smart_ptr< *ExprIsVariant*>

BetterRttiVisitor.**visitExprAsVariant(prog: ProgramPtr; mod: Module?; expr: smart_ptr<ExprAsVariant>) :**

Visitor override that replaces `expr as Type` on Expression subclasses with an RTTI-checked cast via __rtti, panicking on mismatch.

Arguments

- **prog** : *ProgramPtr*
- **mod** : *Module*?
- **expr** : smart_ptr< *ExprAsVariant*>

BetterRttiVisitor.**visitExprSafeAsVariant(prog: ProgramPtr; mod: Module?; expr: smart_ptr<ExprSafeAsVari**

Visitor override that replaces `expr ?as Type` on Expression subclasses with an RTTI-checked cast via __rtti, returning null on mismatch instead of panicking.

Arguments

- **prog** : *ProgramPtr*
- **mod** : *Module*?
- **expr** : smart_ptr< *ExprSafeAsVariant*>

10.3.6 Containers

- *emplace_new (var vec: dasvector`smart_ptr`TypeDecl; var ptr: smart_ptr<TypeDecl>)*
- *emplace_new (var vec: dasvector`smart_ptr`Expression; var ptr: smart_ptr<Expression>)*
- *emplace_new (var vec: MakeStruct; var ptr: smart_ptr<MakeFieldDecl>)*
- *emplace_new (var vec: dasvector`smart_ptr`Variable; var ptr: smart_ptr<Variable>)*

emplace_new

`ast_boost::`**`emplace_new`**`(`*vec: dasvector`smart_ptr`TypeDecl; ptr: smart_ptr<TypeDecl>*`)`

Moves a newly created `smart_ptr` (`Expression`, `TypeDecl`, `Variable`, or `MakeFieldDecl`) into a vector container with correct reference counting.

Arguments

- **vec** : vector<smart_ptr<TypeDecl>>
- **ptr** : smart_ptr< *TypeDecl*>

`ast_boost::`**`emplace_new`**`(`*vec: dasvector`smart_ptr`Expression; ptr: smart_ptr<Expression>*`)`

`ast_boost::`**`emplace_new`**`(`*vec: MakeStruct; ptr: smart_ptr<MakeFieldDecl>*`)`

`ast_boost::`**`emplace_new`**`(`*vec: dasvector`smart_ptr`Variable; ptr: smart_ptr<Variable>*`)`

10.3.7 Textual descriptions of the objects

- *debug_expression (expr: Expression?) : string*
- *debug_expression (expr: ExpressionPtr; deFlags: DebugExpressionFlags = bitfield(0x0)) : string*
- *describe (vvar: VariablePtr) : string*
- *describe (ann: AnnotationDeclaration) : string*
- *describe (expr: Expression?) : string*
- *describe (list: AnnotationArgumentList) : string*
- *describe (list: AnnotationList) : string*
- *describe_bitfield (bf: auto; merger: string = "") : auto*
- *describe_function_short (func: smart_ptr<Function>|Function?) : auto*

debug_expression

`ast_boost::`**`debug_expression(expr: Expression?) : string`**`()`

Returns a hierarchical, Lisp-like textual dump of an `ExpressionPtr` tree showing RTTI types, field values, and nested sub-expressions for debugging.

Arguments

- **expr** : *Expression*?

`ast_boost::`**`debug_expression(expr: ExpressionPtr; deFlags: DebugExpressionFlags = bitfield(0x0)) : strin`**

describe

`ast_boost::`**`describe(vvar: VariablePtr) : string`**`()`

Returns a human-readable textual representation of an AST object (`AnnotationArgumentList`, `AnnotationDeclaration`, `AnnotationList`, `Variable`, or `Expression`).

Arguments

- **vvar** : *VariablePtr*

`ast_boost::`**`describe(ann: AnnotationDeclaration) : string`**`()`

`ast_boost::`**`describe(expr: Expression?) : string`**`()`

`ast_boost::`**`describe(list: AnnotationArgumentList) : string`**`()`

`ast_boost::`**`describe(list: AnnotationList) : string`**`()`

`ast_boost::`**`describe_bitfield(bf: auto; merger: string = "") : auto`**`()`

Returns a textual representation of the set bits in a bitfield value, listing the names of all active flags joined by the specified `merger` string.

Arguments

- **bf** : auto
- **merger** : string

`ast_boost::`**`describe_function_short(func: smart_ptr<Function>|Function?) : auto`**`()`

Returns a compact signature string for a function in the form `name (arg:Type; ...) :  ReturnType`.

Arguments

- **func** : option< *FunctionPtr*| *Function*?>

10.3.8 Queries

- *find_annotation (mod_name: string; ann_name: string) : Annotation const?*
- *find_arg (args: AnnotationArgumentList; argn: string) : RttiValue*
- *find_arg (argn: string; args: AnnotationArgumentList) : RttiValue*
- *find_argument_index (typ: TypeDeclPtr; name: string) : int*
- *find_unique_function (mod: Module?; name: string; canfail: bool = false) : smart_ptr<Function>*
- *find_unique_generic (mod: Module?; name: string; canfail: bool = false) : smart_ptr<Function>*
- *getVectorElementCount (bt: Type) : int*
- *getVectorElementSize (bt: Type) : int*
- *getVectorElementType (bt: Type) : Type*
- *getVectorOffset (bt: Type; ident: string) : int*
- *get_for_source_index (expr: smart_ptr<ExprFor>; svar: VariablePtr) : int*
- *get_for_source_index (expr: smart_ptr<ExprFor>; source: ExpressionPtr) : int*

- *get_workhorse_types () : Type[34]*
- *isCMRES (fun: Function?) : bool*
- *isCMRES (fun: FunctionPtr) : bool*
- *isCMRESType (blockT: TypeDeclPtr) : bool*
- *isExprCallFunc (expr: ExpressionPtr) : bool*
- *isExpression (t: TypeDeclPtr; top: bool = true) : bool*
- *isMakeLocal (expr: ExpressionPtr) : bool*
- *isVectorType (typ: Type) : bool*
- *is_class_method (cinfo: StructurePtr; finfo: TypeDeclPtr) : bool*
- *is_same_or_inherited (parent: Structure const?; child: Structure const?) : bool*

`ast_boost::`**`find_annotation(mod_name: string; ann_name: string) : Annotation const?()`**

Looks up an `Annotation` by name within the specified module during compilation and returns a pointer to it, or `null` if not found.

Arguments

- **mod_name** : string
- **ann_name** : string

find_arg

`ast_boost::`**`find_arg(args: AnnotationArgumentList; argn: string) : RttiValue()`**

Searches an `AnnotationArgumentList` for an argument by name and returns its `RttiValue`; returns `nothing` if the argument is not present.

Arguments

- **args** : *AnnotationArgumentList*
- **argn** : string

`ast_boost::`**`find_arg(argn: string; args: AnnotationArgumentList) : RttiValue()`**

`ast_boost::`**`find_argument_index(typ: TypeDeclPtr; name: string) : int()`**

Searches the `argNames` of a `TypeDeclPtr` (tuple or variant) for the given name and returns its zero-based index, or `-1` if not found.

Arguments

- **typ** : *TypeDeclPtr*
- **name** : string

`ast_boost::`**`find_unique_function(mod: Module?; name: string; canfail: bool = false) : smart_ptr<Function`**

Searches the compiling program for exactly one non-generic function with the given name; returns it or `null` if zero or multiple matches exist.

Arguments

- **mod** : *Module*?
- **name** : string
- **canfail** : bool

`ast_boost::find_unique_generic(mod: Module?; name: string; canfail: bool = false) : smart_ptr<Function>`

Searches the compiling program for exactly one generic function with the given name; returns it or `null` if zero or multiple matches exist.

Arguments

- **mod** : *Module*?
- **name** : string
- **canfail** : bool

`ast_boost::getVectorElementCount(bt: Type) : int()`

Returns the number of scalar elements in a vector `Type` (e.g., 2 for `float2/range`, 3 for `float3`, 4 for `float4`), or 0 for non-vector types.

Arguments

- **bt** : *Type*

`ast_boost::getVectorElementSize(bt: Type) : int()`

Returns the byte size of a single scalar element in a vector `Type` — 8 for `range64/urange64`, 4 for all other vector types.

Arguments

- **bt** : *Type*

`ast_boost::getVectorElementType(bt: Type) : Type()`

Returns the scalar `Type` of each element in a vector type (e.g., `tFloat` for `float2`, `tInt` for `int3` and `range`, `tInt64` for `range64`).

Arguments

- **bt** : *Type*

`ast_boost::getVectorOffset(bt: Type; ident: string) : int()`

Returns the zero-based element index for a named swizzle component (`x/y/z/w` or `r/g/b/a`) within a vector `Type`, or `-1` if out of bounds.

Arguments

- **bt** : *Type*
- **ident** : string

get_for_source_index

ast_boost::**get_for_source_index(expr: smart_ptr<ExprFor>; svar: VariablePtr) : int**()

Returns the zero-based index of a given iterator variable or source expression within a `for` loop's source list, or `-1` if not found.

Arguments

- **expr** : smart_ptr< *ExprFor*>
- **svar** : *VariablePtr*

ast_boost::**get_for_source_index(expr: smart_ptr<ExprFor>; source: ExpressionPtr) : int**()

---

ast_boost::**get_workhorse_types() : Type[34]**()

Returns a fixed array of all commonly used `Type` values — booleans, strings, pointers, numeric scalars, enumerations, bitfields, vectors, and ranges.

isCMRES

ast_boost::**isCMRES(fun: Function?) : bool**()

Returns `true` if a `Function` returns its result via copy-or-move-result-on-stack (CMRES) semantics rather than through a register.

Arguments

- **fun** : *Function*?

ast_boost::**isCMRES(fun: FunctionPtr) : bool**()

---

ast_boost::**isCMRESType(blockT: TypeDeclPtr) : bool**()

Returns `true` if a `TypeDeclPtr` represents a reference type without an explicit `ref` flag, meaning it will use copy-or-move-on-stack semantics.

Arguments

- **blockT** : *TypeDeclPtr*

ast_boost::**isExprCallFunc(expr: ExpressionPtr) : bool**()

Returns `true` if the expression's RTTI tag is `ExprCallFunc`, `ExprOp`, `ExprNew`, or `ExprCall` — i.e., any function-call-like expression.

Arguments

- **expr** : *ExpressionPtr*

ast_boost::**isExpression(t: TypeDeclPtr; top: bool = true) : bool**()

Returns `true` if the given `TypeDeclPtr` refers to an `ast` module handled type whose name starts with `Expr`, including pointer-to-expression types.

Arguments

- **t** : *TypeDeclPtr*

- **top** : bool

ast_boost::**isMakeLocal(expr: ExpressionPtr) : bool**()

Returns `true` if the expression is any `ExprMakeLocal` subclass: `ExprMakeStruct`, `ExprMakeArray`, `ExprMakeTuple`, or `ExprMakeVariant`.

Arguments

- **expr** : *ExpressionPtr*

ast_boost::**isVectorType(typ: Type) : bool**()

Returns `true` if the given `Type` is a vector, range, or urange type (`int2`..``int4``, `uint2`..``uint4``, `float2`..``float4``, `range`, `urange`, `range64`, `urange64`).

Arguments

- **typ** : *Type*

ast_boost::**is_class_method(cinfo: StructurePtr; finfo: TypeDeclPtr) : bool**()

Returns `true` if a `TypeDeclPtr` field represents a class method — a non-dim `tFunction` whose first argument is the class structure (or a parent of it).

Arguments

- **cinfo** : *StructurePtr*
- **finfo** : *TypeDeclPtr*

ast_boost::**is_same_or_inherited(parent: Structure const?; child: Structure const?) : bool**()

Returns `true` if `child` is the same `Structure` as `parent` or is transitively inherited from `parent` by walking the parent chain.

Arguments

- **parent** : *Structure*?
- **child** : *Structure*?

10.3.9 Annotations

- *add_annotation_argument (var arguments: AnnotationArgumentList; argName: string; val: string) : int*
- *add_annotation_argument (var arguments: AnnotationArgumentList; argName: string; val: bool) : int*
- *add_annotation_argument (var arguments: AnnotationArgumentList; argName: string; val: int) : int*
- *add_annotation_argument (var arguments: AnnotationArgumentList; ann: AnnotationArgument) : int*
- *add_annotation_argument (var arguments: AnnotationArgumentList; argName: string; val: float) : int*
- *append_annotation (var blk: smart_ptr<ExprBlock>; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:uint64;tFloat:float;tDouble:dou*
- *append_annotation (var func: FunctionPtr; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:uint64;tFloat:float;tDouble:dou*
- *append_annotation (var st: smart_ptr<Structure>; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:uint64;tFloat:float;tDouble:dou*
- *append_annotation (var blk: smart_ptr<ExprBlock>; mod_name: string; ann_name: string)*
- *append_annotation (mod_name: string; ann_name: string) : smart_ptr<AnnotationDeclaration>*

- *append_annotation (mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:uint64;tFloat:float;tDouble:dou : smart_ptr<AnnotationDeclaration>*
- *append_annotation (var func: FunctionPtr; mod_name: string; ann_name: string)*
- *append_annotation (var st: smart_ptr<Structure>; mod_name: string; ann_name: string)*

add_annotation_argument

`ast_boost::`**`add_annotation_argument(arguments: AnnotationArgumentList; argName: string; val: string) : i`**

Adds a typed annotation argument (`bool`, `int`, `float`, `string`, or `AnnotationArgument`) to an `AnnotationArgumentList` and returns the new argument index.

Arguments

- **arguments** : *AnnotationArgumentList*
- **argName** : string
- **val** : string

`ast_boost::`**`add_annotation_argument(arguments: AnnotationArgumentList; argName: string; val: bool) : int`**

`ast_boost::`**`add_annotation_argument(arguments: AnnotationArgumentList; argName: string; val: int) : int(`**

`ast_boost::`**`add_annotation_argument(arguments: AnnotationArgumentList; ann: AnnotationArgument) : int()`**

`ast_boost::`**`add_annotation_argument(arguments: AnnotationArgumentList; argName: string; val: float) : in`**

---

append_annotation

`ast_boost::`**`append_annotation`**(*blk: smart_ptr<ExprBlock>; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:*

Creates an `AnnotationDeclaration` for the named annotation (with optional typed arguments) and attaches it to a `Function`, `ExprBlock`, or `Structure`.

Arguments

- **blk** : smart_ptr< *ExprBlock*>
- **mod_name** : string
- **ann_name** : string
- **args** : array<tuple<argname:string;argvalue: *RttiValue*>>

`ast_boost::`**`append_annotation`**(*func: FunctionPtr; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:*

`ast_boost::`**`append_annotation`**(*st: smart_ptr<Structure>; mod_name: string; ann_name: string; args: array<tuple<argname:string;argvalue:variant<tBool:bool;tInt:int;tUInt:uint;tInt64:int64;tUInt64:*

`ast_boost::`**`append_annotation`**(*blk: smart_ptr<ExprBlock>; mod_name: string; ann_name: string*)

`ast_boost::`**`append_annotation(mod_name: string; ann_name: string) : smart_ptr<AnnotationDeclaration>()`**

ast_boost::**append_annotation(mod_name: string; ann_name: string; args: array<tuple<argname:string;argva**

ast_boost::**append_annotation**(*func: FunctionPtr; mod_name: string; ann_name: string*)

ast_boost::**append_annotation**(*st: smart_ptr<Structure>; mod_name: string; ann_name: string*)

10.3.10 Expression generation

- *convert_to_expression (var value: auto ==const) : auto*
- *convert_to_expression (value: auto ==const; at: LineInfo) : auto*
- *convert_to_expression (var value: auto& ==const; at: LineInfo) : auto*
- *convert_to_expression (value: auto ==const) : auto*
- *make_static_assert_false (text: string; at: LineInfo) : smart_ptr<ExprStaticAssert>*
- *override_method (var str: StructurePtr; name: string; funcName: string) : bool*
- *panic_expr_as () : void?*

convert_to_expression

ast_boost::**convert_to_expression(value: auto ==const) : auto**()

Converts a runtime value of any supported type to an equivalent AST `ExpressionPtr` that would produce that value when compiled, using `typeinfo` for reflection.

Arguments

- **value** : auto!

ast_boost::**convert_to_expression(value: auto ==const; at: LineInfo) : auto**()

ast_boost::**convert_to_expression(value: auto& ==const; at: LineInfo) : auto**()

ast_boost::**convert_to_expression(value: auto ==const) : auto**()

---

ast_boost::**make_static_assert_false(text: string; at: LineInfo) : smart_ptr<ExprStaticAssert>**()

Creates an `ExprStaticAssert` expression node that always fails at compile time with the given error message text.

Arguments

- **text** : string
- **at** : *LineInfo*

ast_boost::**override_method(str: StructurePtr; name: string; funcName: string) : bool**()

Replaces the initializer of a field named `name` in the given structure with an `@@funcName` function address cast, effectively overriding that class method.

Arguments

- **str** : *StructurePtr*
- **name** : string
- **funcName** : string

ast_boost::**panic_expr_as() : void?**()

Helper function that panics with `"invalid 'as' expression or null pointer dereference"` and returns `null` — used as the failure branch in `as` variant casts.

10.3.11 Visitors

- *visit_finally (blk: ExprBlock?; adapter: smart_ptr<VisitorAdapter>)*

ast_boost::**visit_finally**(*blk: ExprBlock?; adapter: smart_ptr<VisitorAdapter>*)

Invokes the given `VisitorAdapter` on the `finally` section of an `ExprBlock`, allowing macro visitors to inspect or transform finally-block code.

Arguments

- **blk** : *ExprBlock*?
- **adapter** : smart_ptr< *VisitorAdapter*>

10.3.12 Type generation

- *function_to_type (fn: FunctionPtr) : TypeDeclPtr*

ast_boost::**function_to_type(fn: FunctionPtr) : TypeDeclPtr**()

Constructs a `TypeDeclPtr` of `tFunction` base type from a `FunctionPtr`, capturing its argument types and names plus the return type.

Arguments

- **fn** : *FunctionPtr*

10.3.13 Type casts

- *Annotation? `as `FunctionAnnotation (foo: Annotation?) : FunctionAnnotation?*
- *Annotation? `as `StructureAnnotation (foo: Annotation?) : StructureAnnotation?*
- *Annotation? `is `FunctionAnnotation (foo: Annotation?) : bool*
- *Annotation? `is `StructureAnnotation (foo: Annotation?) : bool*
- *Function? `as `BuiltInFunction (foo: Function?) : BuiltInFunction?*
- *Function? `as `ExternalFnBase (foo: Function?) : ExternalFnBase?*
- *Function? `is `BuiltInFunction (foo: Function?) : bool*
- *Function? `is `ExternalFnBase (foo: Function?) : bool*
- *auto `is `BuiltInFunction (anything: auto) : auto*
- *auto `is `ExternalFnBase (anything: auto) : auto*
- *auto `is `FunctionAnnotation (anything: auto) : auto*
- *auto `is `StructureAnnotation (anything: auto) : auto*
- *smart_ptr<Annotation> `as `FunctionAnnotation (foo: smart_ptr<Annotation>) : FunctionAnnotation?*
- *smart_ptr<Annotation> `as `StructureAnnotation (foo: smart_ptr<Annotation>) : StructureAnnotation?*

- *smart_ptr<Annotation> `is`FunctionAnnotation (foo: smart_ptr<Annotation>) : bool*
- *smart_ptr<Annotation> `is`StructureAnnotation (foo: smart_ptr<Annotation>) : bool*
- *walk_and_convert (data: uint8 const?; info: TypeDeclPtr; at: LineInfo) : ExpressionPtr*

ast_boost::**Annotation?`as`FunctionAnnotation(foo: Annotation?) : FunctionAnnotation?**()

Casts an `Annotation?` or `smart_ptr<Annotation>` to `FunctionAnnotation?` via `reinterpret`, verifying the annotation kind first (panics otherwise).

Arguments

- **foo** : *Annotation?*

ast_boost::**Annotation?`as`StructureAnnotation(foo: Annotation?) : StructureAnnotation?**()

Casts an `Annotation?` or `smart_ptr<Annotation>` to `StructureAnnotation?` via `reinterpret`, verifying the annotation kind first (panics otherwise).

Arguments

- **foo** : *Annotation?*

ast_boost::**Annotation?`is`FunctionAnnotation(foo: Annotation?) : bool**()

Returns `true` if the given `Annotation?` or `smart_ptr<Annotation>` is a `FunctionAnnotation` according to its `isFunctionAnnotation` property.

Arguments

- **foo** : *Annotation?*

ast_boost::**Annotation?`is`StructureAnnotation(foo: Annotation?) : bool**()

Returns `true` if the given `Annotation?` or `smart_ptr<Annotation>` is a `StructureAnnotation` according to its `isStructureAnnotation` property.

Arguments

- **foo** : *Annotation?*

ast_boost::**Function?`as`BuiltInFunction(foo: Function?) : BuiltInFunction?**()

Casts a `Function?` to `BuiltInFunction?` via `reinterpret`, verifying the target is a built-in function first (panics otherwise).

Arguments

- **foo** : *Function?*

ast_boost::**Function?`as`ExternalFnBase(foo: Function?) : ExternalFnBase?**()

Casts a `Function?` to `ExternalFnBase?` via `reinterpret`, verifying it is a property-flagged built-in function first (panics otherwise).

Arguments

- **foo** : *Function?*

ast_boost::**Function?`is`BuiltInFunction(foo: Function?) : bool**()

Returns `true` if the given `Function?` has the `builtIn` flag set, indicating it is a `BuiltInFunction`; returns `false` for any other type.

Arguments

- **foo** : *Function*?

ast_boost::**Function?`is`ExternalFnBase(foo: Function?) : bool**()

Returns `true` if the given `Function?` is both `builtIn` and has the `propertyFunction` flag, indicating it is an `ExternalFnBase`; returns `false` otherwise.

Arguments

- **foo** : *Function*?

ast_boost::**auto`is`BuiltInFunction(anything: auto) : auto**()

Returns `true` if the given `Function?` has the `builtIn` flag set, indicating it is a `BuiltInFunction`; returns `false` for any other type.

Arguments

- **anything** : auto

ast_boost::**auto`is`ExternalFnBase(anything: auto) : auto**()

Returns `true` if the given `Function?` is both `builtIn` and has the `propertyFunction` flag, indicating it is an `ExternalFnBase`; returns `false` otherwise.

Arguments

- **anything** : auto

ast_boost::**auto`is`FunctionAnnotation(anything: auto) : auto**()

Returns `true` if the given `Annotation?` or `smart_ptr<Annotation>` is a `FunctionAnnotation` according to its `isFunctionAnnotation` property.

Arguments

- **anything** : auto

ast_boost::**auto`is`StructureAnnotation(anything: auto) : auto**()

Returns `true` if the given `Annotation?` or `smart_ptr<Annotation>` is a `StructureAnnotation` according to its `isStructureAnnotation` property.

Arguments

- **anything** : auto

ast_boost::**smart_ptr<Annotation>`as`FunctionAnnotation(foo: smart_ptr<Annotation>) : FunctionAnnotation**

Casts an `Annotation?` or `smart_ptr<Annotation>` to `FunctionAnnotation?` via `reinterpret`, verifying the annotation kind first (panics otherwise).

Arguments

- **foo** : smart_ptr< *Annotation*>

ast_boost::**smart_ptr<Annotation>`as`StructureAnnotation(foo: smart_ptr<Annotation>) : StructureAnnotati**

Casts an `Annotation?` or `smart_ptr<Annotation>` to `StructureAnnotation?` via `reinterpret`, verifying the annotation kind first (panics otherwise).

Arguments

- **foo** : smart_ptr< *Annotation*>

ast_boost::**smart_ptr<Annotation>`is`FunctionAnnotation(foo: smart_ptr<Annotation>) : bool**()

Returns `true` if the given `Annotation`? or `smart_ptr`<`Annotation`> is a `FunctionAnnotation` according to its `isFunctionAnnotation` property.

Arguments

- **foo** : smart_ptr< *Annotation*>

ast_boost::**smart_ptr<Annotation>`is`StructureAnnotation(foo: smart_ptr<Annotation>) : bool**()

Returns `true` if the given `Annotation`? or `smart_ptr`<`Annotation`> is a `StructureAnnotation` according to its `isStructureAnnotation` property.

Arguments

- **foo** : smart_ptr< *Annotation*>

ast_boost::**walk_and_convert(data: uint8 const?; info: TypeDeclPtr; at: LineInfo) : ExpressionPtr**()

Recursively walks raw data bytes using a `TypeDeclPtr` schema and builds an equivalent AST expression tree that would reproduce that data when compiled.

Arguments

- **data** : uint8?
- **info** : *TypeDeclPtr*
- **at** : *LineInfo*

10.3.14 Setup

- *setup_call_list (name: string; at: LineInfo; isInit: bool = false; isPrivate: bool = true; isLateInit: bool = false) : ExprBlock?*
- *setup_call_list (name: string; at: LineInfo; subblock: block<(var fn:FunctionPtr):void>) : ExprBlock?*
- *setup_macro (name: string; at: LineInfo) : ExprBlock?*
- *setup_tag_annotation (name: string; tag: string; var classPtr: auto) : auto*

setup_call_list

ast_boost::**setup_call_list(name: string; at: LineInfo; isInit: bool = false; isPrivate: bool = true; isL**

Creates or locates a compilation-phase setup function (`__setup_macros`) and returns its body `ExprBlock` so callers can append registration calls to it.

Arguments

- **name** : string
- **at** : *LineInfo*
- **isInit** : bool
- **isPrivate** : bool
- **isLateInit** : bool

ast_boost::**setup_call_list(name: string; at: LineInfo; subblock: block<(var fn:FunctionPtr):void>) : Exp**

---

ast_boost::**setup_macro(name: string; at: LineInfo) : ExprBlock?**()

Creates or locates a macro initialization function (`__setup_macros`) guarded by `is_compiling_macros` and returns its body block for appending macro registration code.

Arguments

- **name** : string
- **at** : *LineInfo*

ast_boost::**setup_tag_annotation(name: string; tag: string; classPtr: auto) : auto**()

Creates an `AstFunctionAnnotation` instance and automatically applies it to every function that carries a matching `[tag_function(tag)]` annotation in the module.

Arguments

- **name** : string
- **tag** : string
- **classPtr** : auto

10.4 decltype macro and template function annotation

The TEMPLATES module implements template instantiation utilities for daslang code generation. It supports stamping out parameterized code patterns with type and value substitution.

See also *Template application helpers* for template substitution and code generation.

All functions and symbols are in "templates" module, use require to get access to it.

```
require daslib/templates
```

10.4.1 Function annotations

templates::**template**

This macro is used to remove unused (template) arguments from the instantiation of the generic function. When [template(x)] is specified, the argument x is removed from the function call, but the type of the instance remains. The call where the function is instanciated is adjusted as well. For example:

```
[template (a), sideeffects]
def boo ( x : int; a : auto(TT) )   // when boo(1,type<int>)
    return "{x}_{typeinfo(typename type<TT>)}"
...
boo(1,type<int>) // will be replaced with boo(1). instace will print "1_int"
```

10.4.2 Call macros

templates::**decltype**

This macro returns ast::TypeDecl for the corresponding expression. For example:

```
let x = 1
let y <- decltype(x) // [[TypeDecl() baseType==Type tInt, flags=TypeDeclFlags constant |␣
↪TypeDeclFlags ref]]
```

templates::**decltype_noref**

This macro returns TypeDecl for the corresponding expression, minus the ref (&) portion.

10.5 Template application helpers

The TEMPLATES_BOOST module extends template utilities with high-level macros for common code generation patterns, including template function generation, type-parameterized struct creation, and compile-time code expansion.

See also *decltype macro and template function annotation* for `decltype` and `[template]` annotations.

All functions and symbols are in "templates_boost" module, use require to get access to it.

```
require daslib/templates_boost
```

10.5.1 Structures

templates_boost::**Template**

This structure contains collection of substitution rules for a template.

Fields

- **kaboomVar** : table<string;tuple<prefix:string;suffix:string>> - variable field access replacement rules
- **call2name** : table<string;string> - call name replacement rules
- **field2name** : table<string;string> - field name replacement rules
- **var2name** : table<string;string> - variable name replacement rules
- **var2expr** : table<string;smart_ptr< *Expression*>> - variable expression replacement rules
- **var2exprList** : table<string;array<smart_ptr< *Expression*>>> - variable expression list replacement rules
- **type2type** : table<string;string> - type name replacement rules
- **type2etype** : table<string; *TypeDeclPtr*> - type to type declaration replacement rules
- **struct2etype** : table< *Structure*?; *TypeDeclPtr*> - structure to type declaration replacement rules
- **blockArgName** : table<string;string> - block argument name replacement rules
- **annArg** : table<string;lambda<(ann: *AnnotationDeclaration*):void>> - annotation argument replacement rules
- **blkArg** : table<string;array< *VariablePtr*>> - block argument replacement rules

- **tag2expr** : table<string;smart_ptr< *Expression*>> - tag expression replacement rules

10.5.2 Call macros

`templates_boost::`**`qmacro_variable`**

This macro implements expression reification for variables.

`templates_boost::`**`qmacro_expr`**

This macro implements first line of the expression block reification 'qmacro_expr'

`templates_boost::`**`qmacro_function`**

This macro implements expression reification for functions.

`templates_boost::`**`qmacro_block_to_array`**

This macro implements expression block to array reification 'qmacro_block_to_array'

`templates_boost::`**`qmacro_template_class`**

Call macro for quoting named template class methods. This macro implements expression reification for the named expressions (function, variable, etc.)

`templates_boost::`**`qmacro_method`**

This macro implements expression reification for class methods.

`templates_boost::`**`qmacro_template_function`**

Call macro for quoting named template functions. This macro implements expression reification for the named expressions (function, variable, etc.)

`templates_boost::`**`qmacro`**

This macro implements expression reification 'qmacro'

`templates_boost::`**`qmacro_block`**

This macro implements expression block reification 'qmacro_block'

`templates_boost::`**`qmacro_type`**

This macro implements type declaration reification 'qmacro_type'

10.5.3 Template rules

- *kaboomVarField (var self: Template; name: string; prefix: string; suffix: string)*
- *renameCall (var self: Template; name: string; newName: string)*
- *renameCall (var self: Template; name: string; newName: das_string)*
- *renameField (var self: Template; name: string; newName: string)*
- *renameField (var self: Template; name: string; newName: das_string)*
- *renameVariable (var self: Template; name: string; newName: string)*
- *renameVariable (var self: Template; name: string; newName: das_string)*

- *replaceAnnotationArgument (var self: Template; name: string; var cb: lambda<(var ann:AnnotationDeclaration):void>)*
- *replaceArgumentWithList (var self: Template; name: string; blka: array<VariablePtr>)*
- *replaceBlockArgument (var self: Template; name: string; newName: string)*
- *replaceStructWithTypeDecl (var self: Template; pstruct: Structure?; var expr: TypeDeclPtr)*
- *replaceType (var self: Template; name: string; newName: string)*
- *replaceTypeWithTypeDecl (var self: Template; name: string; var expr: TypeDeclPtr)*
- *replaceVarTag (var self: Template; name: string; var expr: smart_ptr<Expression>)*
- *replaceVariable (var self: Template; name: string; var expr: smart_ptr<Expression>)*
- *replaceVariableWithList (var self: Template; name: string; expr: dasvector`smart_ptr`Expression)*
- *replaceVariableWithList (var self: Template; name: string; expr: array<ExpressionPtr>)*

`templates_boost::`**`kaboomVarField`**(*self: Template; name: string; prefix: string; suffix: string*)

Adds a rule to to the template to replace a variable field access with a prefix and suffix. I.e. foo.bar into prefix + bar + suffix

Arguments

- **self** : *Template*
- **name** : string
- **prefix** : string
- **suffix** : string

renameCall

`templates_boost::`**`renameCall`**(*self: Template; name: string; newName: string*)

Adds a rule to the template to rename a call.

Arguments

- **self** : *Template*
- **name** : string
- **newName** : string

`templates_boost::`**`renameCall`**(*self: Template; name: string; newName: das_string*)

renameField

`templates_boost::`**`renameField`**`(`*self: Template; name: string; newName: string*`)`

Adds a rule to the template to rename any field lookup (., ?., as, is, etc)

Arguments

- **self** : *Template*
- **name** : string
- **newName** : string

`templates_boost::`**`renameField`**`(`*self: Template; name: string; newName: das_string*`)`

---

renameVariable

`templates_boost::`**`renameVariable`**`(`*self: Template; name: string; newName: string*`)`

Adds a rule to the template to rename a variable.

Arguments

- **self** : *Template*
- **name** : string
- **newName** : string

`templates_boost::`**`renameVariable`**`(`*self: Template; name: string; newName: das_string*`)`

`templates_boost::`**`replaceAnnotationArgument`**`(`*self: Template; name: string; cb: lambda<(var ann:AnnotationDeclaration):void>*`)`

Adds a rule to the template to replace an annotation argument with the result of a callback.

Arguments

- **self** : *Template*
- **name** : string
- **cb** : lambda<(ann: *AnnotationDeclaration*):void>

`templates_boost::`**`replaceArgumentWithList`**`(`*self: Template; name: string; blka: array<VariablePtr>*`)`

Adds a rule to the template to replace a block argument with a list of variables.

Arguments

- **self** : *Template*
- **name** : string
- **blka** : array< *VariablePtr*>

`templates_boost::`**`replaceBlockArgument`**`(`*self: Template; name: string; newName: string*`)`

Adds a rule to the template to rename a block argument.

Arguments

- **self** : *Template*
- **name** : string
- **newName** : string

`templates_boost::`**`replaceStructWithTypeDecl`**`(`*self: Template; pstruct: Structure?; expr: TypeDeclPtr*`)`

Adds a rule to the template to replace a type alias with another type alias, specified by type declaration.

Arguments

- **self** : *Template*
- **pstruct** : *Structure*?
- **expr** : *TypeDeclPtr*

`templates_boost::`**`replaceType`**`(`*self: Template; name: string; newName: string*`)`

Adds a rule to the template to replace a type alias with another type alias, specified by name.

Arguments

- **self** : *Template*
- **name** : string
- **newName** : string

`templates_boost::`**`replaceTypeWithTypeDecl`**`(`*self: Template; name: string; expr: TypeDeclPtr*`)`

Adds a rule to the template to replace a type alias with another type alias, specified by type declaration.

Arguments

- **self** : *Template*
- **name** : string
- **expr** : *TypeDeclPtr*

`templates_boost::`**`replaceVarTag`**`(`*self: Template; name: string; expr: smart_ptr<Expression>*`)`

Adds a rule to the template to replace a variable tag with an expression.

Arguments

- **self** : *Template*
- **name** : string
- **expr** : smart_ptr< *Expression*>

`templates_boost::`**`replaceVariable`**`(`*self: Template; name: string; expr: smart_ptr<Expression>*`)`

Adds a rule to the template to replace a variable with an expression.

Arguments

- **self** : *Template*
- **name** : string

- **expr** : smart_ptr< *Expression*>

replaceVariableWithList

`templates_boost::`**`replaceVariableWithList`**(*self: Template; name: string; expr: dasvector`smart_ptr`Expression*)

Adds a rule to the template to replace a variable with an expression list.

Arguments

- **self** : *Template*
- **name** : string
- **expr** : vector<smart_ptr<Expression>>

`templates_boost::`**`replaceVariableWithList`**(*self: Template; name: string; expr: array<ExpressionPtr>*)

10.5.4 Template application

- *apply_template (var rules: Template; at: LineInfo; var typ: smart_ptr<TypeDecl>; forceAt: bool = true) : TypeDeclPtr*
- *apply_template (var rules: Template; at: LineInfo; var expr: smart_ptr<Expression>; forceAt: bool = true) : ExpressionPtr*
- *apply_template (at: LineInfo; var expr: smart_ptr<Expression>&; blk: block<(var rules:Template):void>) : ExpressionPtr*
- *apply_template (at: LineInfo; var typ: smart_ptr<TypeDecl>&; blk: block<(var rules:Template):void>) : TypeDeclPtr*
- *apply_template (var expr: smart_ptr<Expression>&; blk: block<(var rules:Template):void>) : ExpressionPtr*

apply_template

`templates_boost::`**`apply_template(rules: Template; at: LineInfo; typ: smart_ptr<TypeDecl>; forceAt: bool`**

Applies the template to the given expression. If *forceAt* is set, the resulting expression will have the same line info as 'at'.

Arguments

- **rules** : *Template*
- **at** : *LineInfo*
- **typ** : smart_ptr< *TypeDecl*>
- **forceAt** : bool

`templates_boost::`**`apply_template(rules: Template; at: LineInfo; expr: smart_ptr<Expression>; forceAt: bo`**

`templates_boost::`**`apply_template(at: LineInfo; expr: smart_ptr<Expression>&; blk: block<(var rules:Templ`**

`templates_boost::`**`apply_template(at: LineInfo; typ: smart_ptr<TypeDecl>&; blk: block<(var rules:Template`**

`templates_boost::`**`apply_template(expr: smart_ptr<Expression>&; blk: block<(var rules:Template):void>) :`**

10.5.5 Expression helpers

- *expression_at (var expr: ExpressionPtr; at: LineInfo) : ExpressionPtr*
- *remove_deref (varname: string; var expr: smart_ptr<Expression>)*
- *visit_expression (var expr: ExpressionPtr; var adapter: smart_ptr<VisitorAdapter>)*

templates_boost::**expression_at(expr: ExpressionPtr; at: LineInfo) : ExpressionPtr**()

Force expression location, then return it.

Arguments

- **expr** : *ExpressionPtr*
- **at** : *LineInfo*

templates_boost::**remove_deref**(*varname: string; expr: smart_ptr<Expression>*)

Removes dereferences of the variable *varname* from the expression. This is typically used when replacing 'workhorse' variable with constant.

Arguments

- **varname** : string
- **expr** : smart_ptr< *Expression*>

templates_boost::**visit_expression**(*expr: ExpressionPtr; adapter: smart_ptr<VisitorAdapter>*)

Visits the expression with the given visitor adapter.

Arguments

- **expr** : *ExpressionPtr*
- **adapter** : smart_ptr< *VisitorAdapter*>

10.5.6 Expression generation

- *make_expression_block (var exprs: dasvector`smart_ptr`Expression) : smart_ptr<ExprBlock>*
- *make_expression_block (var exprs: array<ExpressionPtr>) : smart_ptr<ExprBlock>*

make_expression_block

templates_boost::**make_expression_block(exprs: dasvector`smart_ptr`Expression) : smart_ptr<ExprBlock>**()

Create ExprBlock and move all expressions from expr to the list of the block.

Arguments

- **exprs** : vector<smart_ptr<Expression>>

templates_boost::**make_expression_block(exprs: array<ExpressionPtr>) : smart_ptr<ExprBlock>**()

10.5.7 Block helpers

- *move_unquote_block (var expr: ExpressionPtr) : smart_ptr<ExprBlock>*
- *unquote_block (expr: ExpressionPtr) : smart_ptr<ExprBlock>*

templates_boost::**move_unquote_block(expr: ExpressionPtr) : smart_ptr<ExprBlock>**()

Moves the corresponding block subexpression expression from the ExprMakeBlock.

Arguments

- **expr** : *ExpressionPtr*

templates_boost::**unquote_block(expr: ExpressionPtr) : smart_ptr<ExprBlock>**()

Returns the corresponding block subexpression expression from the ExprMakeBlock.

Arguments

- **expr** : *ExpressionPtr*

10.5.8 Global variable helpers

- *add_global_let (mod: Module?; vname: string; vat: LineInfo; var value: ExpressionPtr) : bool*
- *add_global_private_let (mod: Module?; vname: string; vat: LineInfo; var value: ExpressionPtr) : bool*
- *add_global_private_var (mod: Module?; vname: string; vat: LineInfo; var value: ExpressionPtr) : bool*
- *add_global_var (mod: Module?; vname: string; var typ: TypeDeclPtr; vat: LineInfo; priv: bool; blk: block<(var v:VariablePtr):void>) : bool*
- *add_global_var (mod: Module?; vname: string; vat: LineInfo; var value: ExpressionPtr) : bool*
- *add_global_var (mod: Module?; vname: string; var typ: TypeDeclPtr; vat: LineInfo; priv: bool) : bool*

templates_boost::**add_global_let(mod: Module?; vname: string; vat: LineInfo; value: ExpressionPtr) : boo**

Add global variable to the module, given name and initial value. Variable type will be constant.

Arguments

- **mod** : *Module*?
- **vname** : string
- **vat** : *LineInfo*
- **value** : *ExpressionPtr*

templates_boost::**add_global_private_let(mod: Module?; vname: string; vat: LineInfo; value: ExpressionPt**

Add global variable to the module, given name and initial value. It will be private, and type will be constant.

Arguments

- **mod** : *Module*?
- **vname** : string
- **vat** : *LineInfo*
- **value** : *ExpressionPtr*

`templates_boost::add_global_private_var(mod: Module?; vname: string; vat: LineInfo; value: ExpressionPt`

Add global variable to the module, given name and initial value. It will be private.

Arguments

- **mod** : *Module*?
- **vname** : string
- **vat** : *LineInfo*
- **value** : *ExpressionPtr*

add_global_var

`templates_boost::add_global_var(mod: Module?; vname: string; typ: TypeDeclPtr; vat: LineInfo; priv: boo`

Add global variable to the module, given name and type. *priv* specifies if the variable is private to the block.

Arguments

- **mod** : *Module*?
- **vname** : string
- **typ** : *TypeDeclPtr*
- **vat** : *LineInfo*
- **priv** : bool
- **blk** : block<(v: *VariablePtr*):void>

`templates_boost::add_global_var(mod: Module?; vname: string; vat: LineInfo; value: ExpressionPtr) : boo`

`templates_boost::add_global_var(mod: Module?; vname: string; typ: TypeDeclPtr; vat: LineInfo; priv: boo`

10.5.9 Hygienic names

- *make_unique_private_name (prefix: string; vat: LineInfo) : string*

`templates_boost::make_unique_private_name(prefix: string; vat: LineInfo) : string()`

Generates unique private name for the variable, given prefix and line info.

Warning: The assumption is that line info is unique for the context of the unique name generation. If it is not, additional measures must be taken to ensure uniqueness of prefix.

Arguments

- **prefix** : string
- **vat** : *LineInfo*

10.5.10 Quoting macros

- *apply_qblock (var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : ExpressionPtr*
- *apply_qblock_expr (var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : ExpressionPtr*
- *apply_qblock_to_array (var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : array<ExpressionPtr>*
- *apply_qmacro (var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : ExpressionPtr*
- *apply_qmacro_function (fname: string; var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : FunctionPtr*
- *apply_qmacro_method (fname: string; var parent: StructurePtr; var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : FunctionPtr*
- *apply_qmacro_template_class (instance_name: string; var template_type: smart_ptr<TypeDecl>; blk: block<(var rules:Template):void>) : TypeDeclPtr*
- *apply_qmacro_template_function (func: FunctionPtr; blk: block<(var rules:Template):void>) : FunctionPtr*
- *apply_qmacro_variable (vname: string; var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : VariablePtr*
- *apply_qtype (var expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : TypeDeclPtr*

`templates_boost::`**`apply_qblock(expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : Exp`**

Implementation details for the expression reification. This is a block reification.

Arguments

- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

`templates_boost::`**`apply_qblock_expr(expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>)`**

Implementation details for the expression reification. This is a first line of the block as expression reification.

Arguments

- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

`templates_boost::`**`apply_qblock_to_array(expr: smart_ptr<Expression>; blk: block<(var rules:Template):voi`**

Implementation details for the expression reification. This is a block reification.

Arguments

- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

`templates_boost::`**`apply_qmacro(expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : Exp`**

Implementation details for the expression reification.

Arguments

- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qmacro_function(fname: string; expr: smart_ptr<Expression>; blk: block<(var rules**

Implementation details for reification. This is a function generation reification.

Arguments

- **fname** : string
- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qmacro_method(fname: string; parent: StructurePtr; expr: smart_ptr<Expression>;**

Implementation details for reification. This is a class method function generation reification.

Arguments

- **fname** : string
- **parent** : *StructurePtr*
- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qmacro_template_class(instance_name: string; template_type: smart_ptr<TypeDecl>;**

Implementation details for the expression reification. This is a template class instantiation reification.

Arguments

- **instance_name** : string
- **template_type** : smart_ptr< *TypeDecl*>
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qmacro_template_function(func: FunctionPtr; blk: block<(var rules:Template):void**

Applies template rules to a function, cloning it with substituted types.

Arguments

- **func** : *FunctionPtr*
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qmacro_variable(vname: string; expr: smart_ptr<Expression>; blk: block<(var rule**

Implementation details for reification. This is a variable generation reification.

Arguments

- **vname** : string
- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

templates_boost::**apply_qtype(expr: smart_ptr<Expression>; blk: block<(var rules:Template):void>) : Type**

Implementation details for the expression reification. This is a type declaration reification.

Arguments

- **expr** : smart_ptr< *Expression*>
- **blk** : block<(rules: *Template*):void>

10.5.11 Type pointer helpers

- *add_array_ptr_ref (var a: array<smart_ptr<auto(TT)>>) : array<smart_ptr<TT>>*
- *add_type_ptr_ref (var st: StructurePtr; flags: TypeDeclFlags) : TypeDeclPtr*
- *add_type_ptr_ref (var st: EnumerationPtr; flags: TypeDeclFlags) : TypeDeclPtr*
- *add_type_ptr_ref (a: TypeDeclPtr; flags: TypeDeclFlags) : TypeDeclPtr*
- *add_type_ptr_ref (anything: auto(TT); flags: TypeDeclFlags) : TypeDeclPtr*
- *add_type_ptr_ref (var st: Structure?; flags: TypeDeclFlags) : TypeDeclPtr*
- *add_type_ptr_ref (var st: Enumeration?; flags: TypeDeclFlags) : TypeDeclPtr*

templates_boost::**add_array_ptr_ref(a: array<smart_ptr<auto(TT)>>) : array<smart_ptr<TT>>()**

Implementation details for the reification. This adds any array to the rules.

Arguments

- **a** : array<smart_ptr<auto(TT)>>

add_type_ptr_ref

templates_boost::**add_type_ptr_ref(st: StructurePtr; flags: TypeDeclFlags) : TypeDeclPtr()**

Implementation details for the reification. Creates a type declaration from a structure smart pointer.

Arguments

- **st** : *StructurePtr*
- **flags** : *TypeDeclFlags*

templates_boost::**add_type_ptr_ref(st: EnumerationPtr; flags: TypeDeclFlags) : TypeDeclPtr()**

templates_boost::**add_type_ptr_ref(a: TypeDeclPtr; flags: TypeDeclFlags) : TypeDeclPtr()**

templates_boost::**add_type_ptr_ref(anything: auto(TT); flags: TypeDeclFlags) : TypeDeclPtr()**

templates_boost::**add_type_ptr_ref(st: Structure?; flags: TypeDeclFlags) : TypeDeclPtr()**

templates_boost::**add_type_ptr_ref(st: Enumeration?; flags: TypeDeclFlags) : TypeDeclPtr()**

10.5.12 Structure helpers

- *add_structure_field (var cls: StructurePtr; name: string; var t: TypeDeclPtr; var init: ExpressionPtr) : int*

templates_boost::**add_structure_field(cls: StructurePtr; name: string; t: TypeDeclPtr; init: ExpressionP**

Adds a field to the structure.

Arguments

- **cls** : *StructurePtr*
- **name** : string
- **t** : *TypeDeclPtr*
- **init** : *ExpressionPtr*

10.5.13 Class generation

- *enum_class_type (st: auto) : auto*
- *make_class (name: string; var baseClass: StructurePtr; mod: Module?) : smart_ptr<Structure>*
- *make_class (name: string; mod: Module?) : smart_ptr<Structure>*
- *make_class (name: string; var baseClass: Structure?; mod: Module?) : smart_ptr<Structure>*
- *make_class_constructor (cls: StructurePtr; ctor: FunctionPtr) : smart_ptr<Function>*
- *modify_to_class_member (cls: StructurePtr; fun: FunctionPtr; isExplicit: bool; Constant: bool)*

`templates_boost::`**`enum_class_type(st: auto) : auto`**`()`

return underlying type for the enumeration

Arguments

- **st** : auto

make_class

`templates_boost::`**`make_class(name: string; baseClass: StructurePtr; mod: Module?) : smart_ptr<Structure>`**

Creates a class structure derived from baseClass. Adds __rtti, __finalize fields.

Arguments

- **name** : string
- **baseClass** : *StructurePtr*
- **mod** : *Module*?

`templates_boost::`**`make_class(name: string; mod: Module?) : smart_ptr<Structure>`**`()`

`templates_boost::`**`make_class(name: string; baseClass: Structure?; mod: Module?) : smart_ptr<Structure>`**`()`

`templates_boost::`**`make_class_constructor(cls: StructurePtr; ctor: FunctionPtr) : smart_ptr<Function>`**`()`

Adds a class constructor from a constructor function.

Arguments

- **cls** : *StructurePtr*
- **ctor** : *FunctionPtr*

`templates_boost::`**`modify_to_class_member`**(*cls: StructurePtr; fun: FunctionPtr; isExplicit: bool; Constant: bool*)

Modifies function to be a member of a particular class.

Arguments

- **cls** : *StructurePtr*
- **fun** : *FunctionPtr*
- **isExplicit** : bool
- **Constant** : bool

10.6 AST quasiquoting infrastructure

The QUOTE module provides quasiquotation support for AST construction. It allows building AST nodes using daslang syntax with $-prefixed splice points for inserting computed values, making macro writing more readable and less error-prone than manual AST construction.

All functions and symbols are in "quote" module, use require to get access to it.

```
require daslib/quote
```

10.6.1 Structures

quote::**LineInfoInitData**

Initialization data for source line info reconstruction.

Fields

- **fileInfo** : *FileInfo*? - Pointer to the source file info.
- **column** : uint - Column number (1-based).
- **line** : uint - Line number (1-based).
- **last_column** : uint - Last column number of the range.
- **last_line** : uint - Last line number of the range.

quote::**FileInfoInitData**

Initialization data for reconstructing file info.

Fields

- **name** : string - File name string.
- **tabSize** : int - Tab size for the file.

10.6.2 Clone operations

- *clone (var a: dasvector`LineInfo; b: array<LineInfoInitData>)*
- *clone (var a: dasvector`CaptureEntry; b: array<CaptureEntryInitData>)*
- *clone (var args: AnnotationList; var nargs: array<smart_ptr<AnnotationDeclaration>>)*
- *clone (var a: dasvector`EnumEntry; var b: array<EnumEntryInitData>)*
- *clone (var a: AnnotationArgumentList; b: array<AnnotationArgumentInitData>)*
- *clone_file_info (b: FileInfoInitData) : FileInfo?*
- *clone_line_info (var b: LineInfoInitData) : LineInfo*

clone

`quote::`**`clone`**`(`*a: dasvector`LineInfo; b: array<LineInfoInitData>*`)`

Clones an array of LineInfoInitData into a dasvector of LineInfo.

Arguments

- **a** : vector<LineInfo>
- **b** : array< *LineInfoInitData*>

`quote::`**`clone`**`(`*a: dasvector`CaptureEntry; b: array<CaptureEntryInitData>*`)`

`quote::`**`clone`**`(`*args: AnnotationList; nargs: array<smart_ptr<AnnotationDeclaration>>*`)`

`quote::`**`clone`**`(`*a: dasvector`EnumEntry; b: array<EnumEntryInitData>*`)`

`quote::`**`clone`**`(`*a: AnnotationArgumentList; b: array<AnnotationArgumentInitData>*`)`

`quote::`**`clone_file_info(b: FileInfoInitData) : FileInfo?`**`()`

Creates a FileInfo from a FileInfoInitData struct.

Arguments

- **b** : *FileInfoInitData*

`quote::`**`clone_line_info(b: LineInfoInitData) : LineInfo`**`()`

Creates a LineInfo from a LineInfoInitData struct.

Arguments

- **b** : *LineInfoInitData*

10.6.3 Conversion

- *cvt_to_mks (var args: auto) : smart_ptr<MakeStruct>*

`quote::`**`cvt_to_mks(args: auto) : smart_ptr<MakeStruct>`**`()`

Converts an array of arguments into a MakeStruct smart pointer.

Arguments

- **args** : auto

10.7 Boost package for macro manipulations

The MACRO_BOOST module provides utility macros for macro authors, including pattern matching on AST nodes, code generation helpers, and common transformation patterns used when writing compile-time code.

All functions and symbols are in “macro_boost” module, use require to get access to it.

```
require daslib/macro_boost
```

10.7.1 Structures

`macro_boost::`**`CapturedVariable`**

Stored captured variable together with the *ExprVar* which uses it

Fields

- **variable** : *Variable*? - captured variable
- **expression** : *ExprVar*? - expression which uses the variable
- **eref** : bool - this one indicates if its used by reference and does not come from argument. its only used in JIT

10.7.2 Function annotations

`macro_boost::`**`MacroVerifyMacro`**

This macro convert macro_verify(expr,message,prog,at) to the following code::

if !expr
macro_error(prog,at,message) return [[ExpressionPtr]]

10.7.3 Call macros

`macro_boost::`**`return_skip_lockcheck`**

this is similar to regular return <-, but it does not check for locks

10.7.4 Implementation details

- *macro_verify (expr: bool; prog: ProgramPtr; at: LineInfo; message: string)*

`macro_boost::`**`macro_verify`**(*expr: bool; prog: ProgramPtr; at: LineInfo; message: string*)

Same as verify, only the check will produce macro error, followed by return [[ExpressionPtr]]

Arguments

- **expr** : bool
- **prog** : *ProgramPtr*
- **at** : *LineInfo*
- **message** : string

10.7.5 Block analysis

- *capture_block (expr: ExpressionPtr) : array<CapturedVariable>*
- *collect_finally (expr: ExpressionPtr; alwaysFor: bool = false) : array<ExprBlock?>*
- *collect_labels (expr: ExpressionPtr) : array<int>*

`macro_boost::`**`capture_block(expr: ExpressionPtr) : array<CapturedVariable>`**`()`

Collect all captured variables in the expression.

Arguments

- **expr** : *ExpressionPtr*

`macro_boost::`**`collect_finally(expr: ExpressionPtr; alwaysFor: bool = false) : array<ExprBlock?>`**`()`

Collect all finally blocks in the expression. Returns array of ExprBlock? with all the blocks which have *finally* section Does not go into 'make_block' expression, such as *lambda*, or 'block' expressions

Arguments

- **expr** : *ExpressionPtr*
- **alwaysFor** : bool

`macro_boost::`**`collect_labels(expr: ExpressionPtr) : array<int>`**`()`

Collect all labels in the expression. Returns array of integer with label indices Does not go into 'make_block' expression, such as *lambda*, or 'block' expressions

Arguments

- **expr** : *ExpressionPtr*

10.8 Type macro and template structure support

The TYPEMACRO_BOOST module provides infrastructure for defining type macros — custom compile-time type transformations. Type macros allow introducing new type syntax that expands into standard daslang types during compilation.

All functions and symbols are in "typemacro_boost" module, use require to get access to it.

```
require daslib/typemacro_boost
```

10.8.1 Structures

`typemacro_boost::`**`TypeMacroTemplateArgument`**

Holds a type macro template argument with its name and inferred type.

Fields

- **name** : string - Name of the template argument.
- **argument_type** : *TypeDeclPtr* - Declared argument type from the template signature.
- **inferred_type** : *TypeDeclPtr* - Inferred concrete type after template instantiation.

10.8.2 Function annotations

`typemacro_boost::`**`typemacro_function`**

This macro converts function into a type macro.

`typemacro_boost::`**`typemacro_template_function`**

This one converts function into a type macro that uses template arguments. For example [typemacro_template(TFlatHashTable)] def makeFlatHashTable (macroArgument, passArgument : TypeDeclPtr; KeyType, ValueType : TypeDeclPtr; hashFunctionName : string) : TypeDeclPtr { … } We generate the body that handles template argument inference and instantiation.

10.8.3 Structure macros

`typemacro_boost::`**`typemacro_documentation`**

Structure annotation that stores type macro documentation metadata.

`typemacro_boost::`**`typemacro_template`**

Structure annotation that marks a struct as a type macro template instance.

`typemacro_boost::`**`template_structure`**

This macro creates typemacro function and associates it with the structure. It also creates the typemacro_template_function to associate with it. For example:

```
[template_structure(KeyType,ValueType)] struct template TFlatHashTable { ... }
creates:
1) [typemacro_function] def TFlatHashTable (macroArgument, passArgument : TypeDeclPtr;␣
↪KeyType, ValueType : TypeDeclPtr) : TypeDeclPtr {
    return <- make`template`TFlatHashTable(macroArgument, passArgument, KeyType,␣
↪ValueType)
 }
2) [typemacro_template_function(TFlatHashTable)] def make`template`TFlatHashTable␣
↪(macroArgument, passArgument : TypeDeclPtr; KeyType, ValueType : TypeDeclPtr) :␣
↪TypeDeclPtr {
    return <- default<TypeDeclPtr>
 }
```

10.8.4 Enum helpers

- *int64_to_enum (_enu: auto(ET); value: int64) : ET*

`typemacro_boost::`**`int64_to_enum(_enu: auto(ET); value: int64) : ET`**`()`

Converts an int64 value to the specified enum type via reinterpret cast.

Arguments

- **_enu** : auto(ET)
- **value** : int64

10.8.5 Template structure instantiation

- *is_typemacro_template_instance (passArgument: TypeDeclPtr; templateType: TypeDeclPtr; extra: array<tuple<string;string>> = array<tuple<string;string>>()) : bool*
- *make_typemacro_template_instance (instance_type: Structure?; template_type: Structure?; ex: array<tuple<string;string>> = array<tuple<string;string>>())*
- *template_structure_name (base: Structure?; arguments: array<TypeMacroTemplateArgument>; extra: array<tuple<string;string>> = array<tuple<string;string>>()) : string*

typemacro_boost::**is_typemacro_template_instance(passArgument: TypeDeclPtr; templateType: TypeDeclPtr; e**

template instance is determined by having parent == template.parent

Arguments

- **passArgument** : *TypeDeclPtr*
- **templateType** : *TypeDeclPtr*
- **extra** : array<tuple<string;string>>

typemacro_boost::**make_typemacro_template_instance**(*instance_type: Structure?; template_type: Structure?; ex: array<tuple<string;string>> = array<tuple<string;string>>()*)

Annotates a structure as a typemacro template instance of the given template type.

Arguments

- **instance_type** : *Structure*?
- **template_type** : *Structure*?
- **ex** : array<tuple<string;string>>

typemacro_boost::**template_structure_name(base: Structure?; arguments: array<TypeMacroTemplateArgument>;**

Builds a mangled template structure name from its base name and argument types.

Arguments

- **base** : *Structure*?
- **arguments** : array< *TypeMacroTemplateArgument*>
- **extra** : array<tuple<string;string>>

10.8.6 Type inference helpers

- *add_structure_aliases (structType: Structure?; var args: array<TypeMacroTemplateArgument>)*
- *infer_struct_aliases (structType: Structure?; var args: array<TypeMacroTemplateArgument>) : bool*
- *infer_template_types (passArgument: TypeDeclPtr; var args: array<TypeMacroTemplateArgument>) : TypeDeclPtr*
- *verify_arguments (var args: array<TypeMacroTemplateArgument>) : bool*

typemacro_boost::**add_structure_aliases**(*structType: Structure?; args: array<TypeMacroTemplateArgument>*)

Adds all template argument type aliases to a structure.

Arguments

- **structType** : *Structure*?
- **args** : array< *TypeMacroTemplateArgument*>

typemacro_boost::**infer_struct_aliases(structType: Structure?; args: array<TypeMacroTemplateArgument>) :**

Infers structure alias types for all template arguments from a structure definition.

Arguments

- **structType** : *Structure*?
- **args** : array< *TypeMacroTemplateArgument*>

typemacro_boost::**infer_template_types(passArgument: TypeDeclPtr; args: array<TypeMacroTemplateArgument>**

Infers and validates template argument types against a pass argument, returning the resolved type.

Arguments

- **passArgument** : *TypeDeclPtr*
- **args** : array< *TypeMacroTemplateArgument*>

typemacro_boost::**verify_arguments(args: array<TypeMacroTemplateArgument>) : bool**()

Verifies that all template arguments have been fully inferred (no remaining auto or alias types).

Arguments

- **args** : array< *TypeMacroTemplateArgument*>

10.8.7 String constant access

- *get_string_const (expr: ExpressionPtr) : string*

typemacro_boost::**get_string_const(expr: ExpressionPtr) : string**()

Extracts a string constant value or function address name from an expression.

Arguments

- **expr** : *ExpressionPtr*

10.8.8 Work tracking

- *is_custom_work_done (structType: Structure?) : bool*
- *mark_custom_work_done (var structType: Structure?)*

typemacro_boost::**is_custom_work_done(structType: Structure?) : bool**()

Returns true if custom work has already been performed on the template structure.

Arguments

- **structType** : *Structure*?

`typemacro_boost::`**`mark_custom_work_done`**(*structType: Structure?*)

Marks the template structure's custom work as complete in its annotation.

Arguments

- **structType** : *Structure*?

10.8.9 Type macro arguments

- *typemacro_argument (dimExpr: auto; index: int; var constType: ExprConstString; var defaultValue: auto(ValueT)) : ValueT*
- *typemacro_argument (dimExpr: auto; index: int; var constType: auto(ExprConstType); var defaultValue: auto(ValueT)) : ValueT*

typemacro_argument

`typemacro_boost::`**`typemacro_argument(dimExpr: auto; index: int; constType: ExprConstString; defaultValue`**

Extracts a string constant or function address argument at the given index from a type macro's dimension expressions.

Arguments

- **dimExpr** : auto
- **index** : int
- **constType** : *ExprConstString*
- **defaultValue** : auto(ValueT)

`typemacro_boost::`**`typemacro_argument(dimExpr: auto; index: int; constType: auto(ExprConstType); defaultV`**

10.9 DECS, AST block to loop

The AST_BLOCK_TO_LOOP module provides an AST transformation macro that converts block-based iteration patterns into explicit loop constructs. Used internally by other macro libraries for optimization.

All functions and symbols are in "ast_block_to_loop" module, use require to get access to it.

```
require daslib/ast_block_to_loop
```

10.9.1 Block to loop conversion

- *convert_block_to_loop (var blk: smart_ptr<Expression>; failOnReturn: bool; replaceReturnWithContinue: bool; requireContinueCond: bool)*

`ast_block_to_loop::`**`convert_block_to_loop`**(*blk: smart_ptr<Expression>; failOnReturn: bool; replaceReturnWithContinue: bool; requireContinueCond: bool*)

Converts closure block to loop. If *failOnReturn* is true, then returns are not allowed inside the block. If *replaceReturnWithContinue* is true, then *return cond;* are replaced with *if cond; continue;*. If *requireContinueCond* is false, then *return;* is replaced with *continue;*, otherwise it is an error.

Arguments

- **blk** : smart_ptr< *Expression*>
- **failOnReturn** : bool
- **replaceReturnWithContinue** : bool
- **requireContinueCond** : bool

10.10 AST type usage collection

The AST_USED module implements analysis passes that determine which AST nodes are actually used in the program. This information is used for dead code elimination, tree shaking, and optimizing generated output.

All functions and symbols are in "ast_used" module, use require to get access to it.

```
require daslib/ast_used
```

10.10.1 Structures

`ast_used::`**`OnlyUsedTypes`**

Collection of all structure and enumeration types that are used in the AST.

Fields

- **st** : table< *Structure*?;void> - all structure types used
- **en** : table< *Enumeration*?;void> - all enumeration types used

10.10.2 Collecting type information

- *collect_used_types (vfun: array<Function?>; vvar: array<Variable?>; blk: block<(usedTypes:OnlyUsedTypes):void>)*

`ast_used::`**`collect_used_types`**`(`*vfun: array<Function?>; vvar: array<Variable?>; blk: block<(usedTypes:OnlyUsedTypes):void>*`)`

Goes through list of functions *vfun* and variables *vvar* and collects list of which enumeration and structure types are used in them. Calls *blk* with said list.

Arguments

- **vfun** : array< *Function*?>
- **vvar** : array< *Variable*?>
- **blk** : block<(usedTypes: *OnlyUsedTypes*):void>

10.11 Constant expression checker and substitution

The CONSTANT_EXPRESSION module provides the `[constant_expression]` function annotation. Functions marked with this annotation are evaluated at compile time when all arguments are constants, replacing the call with the computed result.

All functions and symbols are in "constant_expression" module, use require to get access to it.

```
require daslib/constant_expression
```

10.11.1 Function annotations

constant_expression::**constexpr**

This macro implements a constexpr function argument checker. Given list of arguments to verify, it will fail for every one where non-constant expression is passed. For example:

```
[constexpr (a)]
def foo ( t:string; a : int )
    print("{t} = {a}\n")
var BOO = 13
[export]
def main
    foo("blah", 1)
    foo("ouch", BOO)    // compilation error: `a is not a constexpr, BOO`
```

constant_expression::**constant_expression**

This function annotation implements constant expression folding for the given arguments. When argument is specified in the annotation, and is passed as a constant expression, custom version of the function is generated, and an argument is substituted with a constant value. This allows using of static_if expression on the said arguments, as well as other optimizations. For example:

```
[constant_expression(constString)]
def take_const_arg(constString:string)
    print("constant string is = {constString}\n")   // note - constString here is not an␣
↪argument
```

10.11.2 Macro helpers

- *isConstantExpression (expr: ExpressionPtr) : bool*

constant_expression::**isConstantExpression(expr: ExpressionPtr) : bool**()

This macro function returns true if the expression is a constant expression

Arguments

- **expr** : *ExpressionPtr*

CHAPTER

ELEVEN

ANNOTATIONS AND CONTRACTS

Function annotations, compile-time contracts, type traits, deferred execution, and other compile-time utilities.

11.1 Miscellaneous contract annotations

The CONTRACTS module provides compile-time type constraints for generic function arguments. Annotations like `[expect_any_array]`, `[expect_any_enum]`, `[expect_any_numeric]`, and `[expect_any_struct]` restrict which types can instantiate a generic parameter, producing clear error messages on mismatch.

See tutorial_contracts for a hands-on tutorial.

All functions and symbols are in "contracts" module, use require to get access to it.

```
require daslib/contracts
```

Example:

```
require daslib/contracts

    [!expect_dim(a)]
    def process(a) {
        return "scalar"
    }

    [expect_dim(a)]
    def process(a) {
        return "array"
    }

    [export]
    def main() {
        var arr : int[3]
        print("{process(42)}\n")
        print("{process(arr)}\n")
    }
    // output:
    // scalar
    // array
```

11.1.1 Function annotations

`contracts::`**`expect_any_array`**

[expect_any_array(argname)] contract, which only accepts array<T>, T[], or das`vector<T>

`contracts::`**`expect_any_enum`**

[expect_any_enum(argname)] contract, which only accepts enumerations

`contracts::`**`expect_any_bitfield`**

[expect_any_bitfield(argname)] contract, which only accepts bitfields

`contracts::`**`expect_any_vector_type`**

[expect_any_vector_type(argname)] contract, which only accepts vector types, i.e. int2, float3, range, etc

`contracts::`**`expect_any_struct`**

[expect_any_struct(argname)] contract, which only accepts structs (but not classes)

`contracts::`**`expect_any_numeric`**

[expect_any_numeric(argname)] contract, which only accepts numeric types (int, float, etc)

`contracts::`**`expect_any_workhorse`**

[expect_any_workhorse(argname)] contract, which only accepts workhorse types (int, float, etc) Workhorse types are: bool,int*,uint*,float*,double,range and urange, range64 and urange64, string,enumeration,and non-smart pointers

`contracts::`**`expect_any_workhorse_raw`**

[expect_any_workhorse_raw(argname)] contract, which only accepts workhorse types which are raw (not pointer or bool)

`contracts::`**`expect_any_tuple`**

[expect_any_tuple(argname)] contract, which only accepts tuples

`contracts::`**`expect_any_variant`**

[expect_any_variant(argname)] contract, which only accepts variants

`contracts::`**`expect_any_function`**

[expect_any_function(argname)] contract, which only accepts functions

`contracts::`**`expect_any_lambda`**

[expect_any_lambda(argname)] contract, which only accepts lambdas

`contracts::`**`expect_ref`**

[expect_ref(argname)] contract, which only accepts references

`contracts::`**`expect_pointer`**

[expect_pointer(argname)] contract, which only accepts pointers

`contracts::`**`expect_class`**

[expect_class(argname)] contract, which only accepts class instances

`contracts::`**`expect_value_handle`**

[expect_value_handle(argname)] contract, which only accepts value handles

11.1.2 Type queries

- *isYetAnotherVectorTemplate (td: TypeDeclPtr) : bool*

contracts::**isYetAnotherVectorTemplate(td: TypeDeclPtr) : bool**()

returns true if the given type declaration is a das::vector template bound on C++ side

Arguments

- **td** : *TypeDeclPtr*

11.2 Apply reflection pattern

The APPLY module provides the `apply` macro for iterating over struct, tuple, and variant fields at compile time. Each field is visited with its name and a reference to its value, enabling generic per-field operations like serialization, printing, and validation.

All functions and symbols are in “apply” module, use require to get access to it.

```
require daslib/apply
```

Example:

```
require daslib/apply

    struct Foo {
        a : int
        b : float
        c : string
    }

    [export]
    def main() {
        var foo = Foo(a = 42, b = 3.14, c = "hello")
        apply(foo) $(name, field) {
            print("{name} = {field}\n")
        }
    }
    // output:
    // a = 42
    // b = 3.14
    // c = hello
```

See also:

tutorial_apply — comprehensive tutorial covering structs, tuples, variants, `static_if` dispatch, mutation, generic `describe`, and the 3-argument annotation form.

11.2.1 Call macros

apply::**apply**

This macro implements the apply() pattern. For each field in a structure, variant, or tuple, the block is invoked with the field name and value. An optional third block argument receives per-field annotations as array<tuple<name:string; data:RttiValue>>.

```
struct Bar {
    x, y : float
}
apply(Bar(x=1.0, y=2.0)) $(name, field) {
    print("{name} = {field} ")
}
```

Would print `x = 1 y = 2`

11.3 defer and defer_delete macros

The DEFER module implements the defer pattern — the ability to schedule cleanup code to run at scope exit, similar to Go's defer. The deferred block is moved to the finally section of the enclosing scope at compile time.

All functions and symbols are in "defer" module, use require to get access to it.

```
require daslib/defer
```

Example:

```
require daslib/defer

    [export]
    def main() {
        print("start\n")
        defer() {
            print("cleanup runs last\n")
        }
        print("middle\n")
    }
    // output:
    // start
    // middle
    // cleanup runs last
```

11.3.1 Function annotations

defer::**DeferMacro**

This macro converts defer() <| block expression into {}, and move block to the finally section of the current block

11.3.2 Call macros

defer::**defer_delete**

This macro converts defer_delete() expression into {}, and add delete expression to the finally section of the current block

11.3.3 Defer

- *defer (blk: block<():void>)*

defer::**defer**(*blk: block<():void>*)

defer a block of code. For example:

```
var a = fopen("filename.txt","r")
defer <|
    fclose(a)
```

Will close the file when 'a' is out of scope.

Arguments

- **blk** : block<void>

11.3.4 Stub

- *nada ()*

defer::**nada**()

helper function which does nothing and will be optimized out

11.4 if_not_null macro

The IF_NOT_NULL module provides a null-safe call macro. The expression `ptr |> if_not_null <| call(args)` expands to a null check followed by a dereferenced call: `if (ptr != null) { call(*ptr, args) }`.

All functions and symbols are in "if_not_null" module, use require to get access to it.

```
require daslib/if_not_null
```

11.4.1 Call macros

if_not_null::**if_not_null**

This macro transforms:

```
ptr |> if_not_null <| call(...)
```

to:

```
var _ptr_var = ptr
if _ptr_var
    call(*_ptr_var,...)
```

11.5 is_local_xxx ast helpers

The IS_LOCAL module provides compile-time checks for whether a variable is locally allocated (on the stack) versus heap-allocated. This enables writing generic code that optimizes differently based on allocation strategy.

All functions and symbols are in "is_local" module, use require to get access to it.

```
require daslib/is_local
```

11.5.1 Scope checks

- *is_local_expr (expr: ExpressionPtr) : bool*
- *is_local_or_global_expr (expr: ExpressionPtr) : bool*
- *is_scope_expr (expr: ExpressionPtr) : bool*
- *is_shared_expr (expr: ExpressionPtr) : bool*
- *is_temp_safe (expr: ExpressionPtr) : bool*

is_local::**is_local_expr(expr: ExpressionPtr) : bool**()

Returns true if the expression is local to the current scope.

Arguments

- **expr** : *ExpressionPtr*

is_local::**is_local_or_global_expr(expr: ExpressionPtr) : bool**()

Returns true if expression is local to the current scope or global scope.

Arguments

- **expr** : *ExpressionPtr*

is_local::**is_scope_expr(expr: ExpressionPtr) : bool**()

Returns true if the expression is a scoped expression, i.e. eventually points to a variable.

Arguments

- **expr** : *ExpressionPtr*

is_local::**is_shared_expr(expr: ExpressionPtr) : bool**()

Returns true if the expression refers to a global shared variable.

Arguments

- **expr** : *ExpressionPtr*

is_local::**is_temp_safe(expr: ExpressionPtr) : bool**()

Returns true if the expression had no calls, [] or table [] operators of any kind. This is used to check expression can be safely casted to temp type.

Arguments

- **expr** : *ExpressionPtr*

11.6 safe_addr macro

The SAFE_ADDR module provides compile-time checked pointer operations. `safe_addr` returns a temporary pointer to a variable only if the compiler can verify the pointer will not outlive its target. This prevents dangling pointer bugs without runtime overhead.

All functions and symbols are in "safe_addr" module, use require to get access to it.

```
require daslib/safe_addr
```

11.6.1 Function annotations

safe_addr::**SafeAddrMacro**

This macro reports an error if safe_addr is attempted on the object, which is not local to the scope. I.e. if the object can *expire* while in scope, with delete, garbage collection, or on the C++ side.

safe_addr::**SharedAddrMacro**

This macro reports an error if shared_addr is attempted on anything other that shared global variables. I.e. only global variables are safe to use with shared_addr.

safe_addr::**TempValueMacro**

This macro reports an error if temp_value is attempted outside of function arguments.

11.6.2 Safe temporary address

- *safe_addr (x: auto(T) const& ==const) : T?#*
- *safe_addr (var x: auto(T)& ==const) : T?#*
- *shared_addr (val: auto(VALUE)) : auto*
- *shared_addr (tab: table<auto(KEY), auto(VAL)>; k: KEY) : auto*

safe_addr

`safe_addr::`**`safe_addr(x: auto(T) const& ==const) : T?#()`**

returns temporary pointer to the given expression

Arguments

- **x** : auto(T)&!

`safe_addr::`**`safe_addr(x: auto(T)& ==const) : T?#()`**

---

shared_addr

`safe_addr::`**`shared_addr(val: auto(VALUE)) : auto()`**

returns address of the given shared variable. it's safe because shared variables never go out of scope

Arguments

- **val** : auto(VALUE)&

`safe_addr::`**`shared_addr(tab: table<auto(KEY), auto(VAL)>; k: KEY) : auto()`**

11.6.3 Temporary pointers

- *temp_ptr (var x: auto(T)? implicit ==const) : T?#*
- *temp_ptr (x: auto(T)? const implicit ==const) : T?#*

temp_ptr

`safe_addr::`**`temp_ptr(x: auto(T)? implicit ==const) : T?#()`**

returns temporary pointer from a given pointer

Arguments

- **x** : auto(T)? implicit!

`safe_addr::`**`temp_ptr(x: auto(T)? const implicit ==const) : T?#()`**

11.6.4 Temporary values

- *temp_value (var x: auto(T)& ==const) : T&#*
- *temp_value (x: auto(T) const& ==const) : T const&#*

temp_value

safe_addr::**temp_value(x: auto(T)& ==const) : T&#**()

returns temporary reference to the given expression

Arguments

- **x** : auto(T)&!

safe_addr::**temp_value(x: auto(T) const& ==const) : T const&#**()

11.7 static_let macro

The STATIC_LET module implements the `static_let` pattern — local variables that persist across function calls, similar to C `static` variables. The variable is initialized once on first call and retains its value in subsequent invocations.

All functions and symbols are in "static_let" module, use require to get access to it.

```
require daslib/static_let
```

Example:

```
require daslib/static_let

    def counter() : int {
        static_let() {
            var count = 0
        }
        count ++
        return count
    }

    [export]
    def main() {
        print("{counter()}\n")
        print("{counter()}\n")
        print("{counter()}\n")
    }
    // output:
    // 1
    // 2
    // 3
```

11.7.1 Function annotations

`static_let::`**`StaticLetMacro`**

This macro implements the *static_let* and *static_let_finalize* functions.

11.7.2 Static variable declarations

- *static_let (name: string; blk: block<():void>)*
- *static_let (blk: block<():void>)*
- *static_let_finalize (blk: block<():void>)*

static_let

`static_let::`**`static_let`**`(`*name: string; blk: block<():void>*`)`

Given a scope with the variable declarations, this function will make those variables global. Variable will be renamed under the hood, and all local access to it will be renamed as well.

Arguments

- **name** : string
- **blk** : block<void>

`static_let::`**`static_let`**`(`*blk: block<():void>*`)`

`static_let::`**`static_let_finalize`**`(`*blk: block<():void>*`)`

This is very similar to regular static_let, but additionally the variable will be deleted on the context shutdown.

Arguments

- **blk** : block<void>

11.8 lpipe macro

The LPIPE module provides the `lpipe` macro for passing multiple block arguments to a single function call. While `<|` handles the first block argument, `lpipe` adds subsequent blocks on following lines.

All functions and symbols are in "lpipe" module, use require to get access to it.

```
require daslib/lpipe
```

Example:

```
require daslib/lpipe

    def take2(a, b : block) {
        invoke(a)
        invoke(b)
    }
```

(continues on next page)

(continued from previous page)

```
    [export]
    def main() {
        take2() {
            print("first\n")
        }
        lpipe() {
            print("second\n")
        }
    }
    // output:
    // first
    // second
```

11.8.1 Call macros

lpipe::**lpipe**

This macro will implement the lpipe function. It allows piping blocks the previous line call. For example:

```
def take2(a,b:block)
    invoke(a)
    invoke(b)
...
take2 <|
    print("block1\n")
lpipe <|    // this block will pipe into take2
    print("block2\n")
```

11.9 Assert once

The ASSERT_ONCE module provides the `assert_once` macro — an assertion that triggers only on its first failure. Subsequent failures at the same location are silently ignored, preventing assertion storms in loops or frequently called code.

All functions and symbols are in “assert_once” module, use require to get access to it.

```
require daslib/assert_once
```

11.9.1 Function annotations

assert_once::**AssertOnceMacro**

This macro convert assert_once(expr,message) to the following code:

```
var __assert_once_I = true  // this is a global variable
if __assert_once_I && !expr
    __assert_once_I = false
    assert(false,message)
```

11.9.2 Assertion

- *assert_once (expr: bool; message: string = "")*

`assert_once::`**`assert_once`**(*expr: bool; message: string = ""*)

Same as assert, only the check will be not be repeated after the assertion failed the first time.

Arguments

- **expr** : bool
- **message** : string

11.10 Loop unrolling

The UNROLL module implements compile-time loop unrolling. The `unroll` macro replaces a `for` loop with a constant `range` bound by stamping out each iteration as separate inlined code, eliminating loop overhead.

All functions and symbols are in "unroll" module, use require to get access to it.

```
require daslib/unroll
```

Example:

```
require daslib/unroll

    [export]
    def main() {
        unroll() {
            for (i in range(4)) {
                print("step {i}\n")
            }
        }
    }
    // output:
    // step 0
    // step 1
    // step 2
    // step 3
```

11.10.1 Function annotations

`unroll::`**`UnrollMacro`**

This macro implements loop unrolling in the form of *unroll* function. Unroll function expects block with the single for loop in it. Moveover only range for is supported, and only with the fixed range. For example::

```
var n : float4[9]
unroll <|   // contents of the loop will be replaced with 9 image load instructions.
    for i in range(9)
        n[i] = imageLoad(c_bloom_htex, xy + int2(0,i-4))
```

11.10.2 Unrolling

- *unroll (blk: block<():void>)*

`unroll::`**`unroll`**`(`*blk: block<():void>*`)`

Unrolls the for loop (with fixed range)

Arguments

- **blk** : block<void>

11.11 Bitfield operator overloads

The BITFIELD_BOOST module provides utility macros for working with bitfield types including conversion between bitfield values and strings, and iteration over set bits.

All functions and symbols are in "bitfield_boost" module, use require to get access to it.

```
require daslib/bitfield_boost
```

11.11.1 Bitfield element access

- *auto(TT)&[]&&= (var b: auto(TT)&; i: int; v: bool) : auto*
- *auto(TT)&[]= (var b: auto(TT)&; i: int; v: bool) : auto*
- *auto(TT)&[]^^= (var b: auto(TT)&; i: int; v: bool) : auto*
- *auto(TT)&[]||= (var b: auto(TT)&; i: int; v: bool) : auto*
- *auto[] (b: auto; i: int) : bool*

`bitfield_boost::`**`auto(TT)&[]&&=(b: auto(TT)&; i: int; v: bool) : auto`**`()`

&& assignment for bitfield bit at index i

Arguments

- **b** : auto(TT)&
- **i** : int
- **v** : bool

`bitfield_boost::`**`auto(TT)&[]=(b: auto(TT)&; i: int; v: bool) : auto`**`()`

set bitfield bit at index i to v

Arguments

- **b** : auto(TT)&
- **i** : int
- **v** : bool

`bitfield_boost::`**`auto(TT)&[]^^=(b: auto(TT)&; i: int; v: bool) : auto`**`()`

toggle bitfield bit at index i if v is true

Arguments

- **b** : auto(TT)&
- **i** : int
- **v** : bool

`bitfield_boost::auto(TT)&[]||=(b: auto(TT)&; i: int; v: bool) : auto()`

|| assignment for bitfield bit at index i

Arguments

- **b** : auto(TT)&
- **i** : int
- **v** : bool

`bitfield_boost::auto[](b: auto; i: int) : bool()`

get bitfield bit at index i

Arguments

- **b** : auto
- **i** : int

11.11.2 Iteration

- *each_bit (b: auto) : iterator<bool>*

`bitfield_boost::each_bit(b: auto) : iterator<bool>()`

Iterates over each bit of a bitfield value, yielding true or false for each bit.

Arguments

- **b** : auto

11.12 Bitfield name traits

The BITFIELD_TRAIT module implements reflection utilities for bitfield types: converting bitfield values to and from human-readable strings, iterating over individual set bits, and constructing bitfield values from string names.

All functions and symbols are in "bitfield_trait" module, use require to get access to it.

```
require daslib/bitfield_trait
```

11.12.1 Function annotations

`bitfield_trait::EachBitfieldMacro`

This macro converts each(bitfield) to the following code::

generator<string>() <|
yield field1 yield field2 … return false

`bitfield_trait::`**`EachBitNameBitfieldMacro`**

This macro converts each(bitfield) to the following code::

> **generator<string>() <|**
> > yield “field1” yield “field2” … return false

11.12.2 Iteration

- *each (argT: auto) : auto*
- *each_bit_name (argT: auto) : auto*

`bitfield_trait::`**`each(argT: auto) : auto`**`()`

Iterates over the names of a bitfield type, yielding each bit as a bitfield value (1ul << bitIndex).

> **Arguments**
> - **argT** : auto

`bitfield_trait::`**`each_bit_name(argT: auto) : auto`**`()`

Iterates over the names of a bitfield type, yielding each bit name as a string.

> **Arguments**
> - **argT** : auto

11.13 Enumeration traits

The ENUM_TRAIT module provides reflection utilities for enumerations: iterating over all values, converting between enum values and strings, and building lookup tables. The `[string_to_enum]` annotation generates a string constructor for the annotated enum type.

All functions and symbols are in “enum_trait” module, use require to get access to it.

```
require daslib/enum_trait
```

Example:

```
require daslib/enum_trait

    enum Color {
        red
        green
        blue
    }

    [export]
    def main() {
        print("{Color.green}\n")
        let c = to_enum(type<Color>, "blue")
        print("{c}\n")
        let bad = to_enum(type<Color>, "purple", Color.red)
        print("fallback = {bad}\n")
```

(continues on next page)

(continued from previous page)

```
    }
    // output:
    // green
    // blue
    // fallback = red
```

11.13.1 Typeinfo macros

enum_trait::**enum_names**

Implements typeinfo(enum_names EnumOrEnumType) which returns array of strings with enumValue names.

enum_trait::**enum_length**

Implements typeinfo(enum_length EnumOrEnumType) which returns total number of elements in enumeration.

11.13.2 Handled enumerations

enum_trait::**string_to_enum**

Enumeration annotation which implements string constructor for enumeration.

11.13.3 Enumeration iteration

- *each (tt: auto(TT)) : iterator<TT>*

enum_trait::**each(tt: auto(TT)) : iterator<TT>**()

Returns an iterator over all values of the given enumeration type.

Arguments

- **tt** : auto(TT)

11.13.4 Enumeration conversion

- *enum_to_table (ent: auto(EnumT)) : table<string, EnumT>*
- *string (arg: auto) : auto*
- *to_enum (ent: auto(EnumT); name: string) : EnumT*
- *to_enum (ent: auto(EnumT); name: string; defaultValue: EnumT) : EnumT*

enum_trait::**enum_to_table(ent: auto(EnumT)) : table<string, EnumT>**()

converts enum type to a table of name => value pairs
usage: let t = enum_to_table(type<EnumType>)

Arguments

- **ent** : auto(EnumT)

enum_trait::**string(arg: auto) : auto**()

converts enum value to string

usage: let s = string(EnumValue)

Arguments

- **arg** : auto

to_enum

enum_trait::**to_enum(ent: auto(EnumT); name: string) : EnumT**()

converts string to enum value, panics if not found

usage: let e = to_enum(type<EnumType>,"EnumValueName")

Arguments

- **ent** : auto(EnumT)
- **name** : string

enum_trait::**to_enum(ent: auto(EnumT); name: string; defaultValue: EnumT) : EnumT**()

11.14 Type trait macros

The TYPE_TRAITS module provides compile-time type introspection and manipulation. It includes type queries (`is_numeric`, `is_string`, `is_pointer`), type transformations, and generic programming utilities for writing type-aware macros and functions.

All functions and symbols are in "type_traits" module, use require to get access to it.

```
require daslib/type_traits
```

11.14.1 Call macros

type_traits::**is_subclass_of**

Converts to 'true' if the first type is a subclass of the second type.

11.14.2 Typeinfo macros

type_traits::**safe_has_property**

this macro implements "has_property" type trait, which returns true when structure has a property

type_traits::**fields_count**

this macro implements "fields_count" type trait, which returns total number of fields in the structure

11.15 C++ bindings generator

The CPP_BIND module provides utilities for generating daslang bindings to C++ code. It helps generate module registration code, type annotations, and function wrappers for exposing C++ APIs to daslang programs.

All functions and symbols are in "cpp_bind" module, use require to get access to it.

```
require daslib/cpp_bind
```

11.15.1 Generation of bindings

- *log_cpp_class_adapter (cpp_file: file; name: string; cinfo: TypeDeclPtr)*

cpp_bind::**log_cpp_class_adapter**(*cpp_file: file; name: string; cinfo: TypeDeclPtr*)

Generates C++ class adapter for the Daslang class. Intended use:

```
log_cpp_class_adapter(cppFileNameDotInc, "daslangClassName", typeinfo(ast_typedecl type
↪<daslangClassName>))
```

Arguments

- **cpp_file** : *file*
- **name** : string
- **cinfo** : *TypeDeclPtr*

CHAPTER

TWELVE

CLASSES AND INTERFACES

Class infrastructure, interface definitions, dynamic casting, and constructor/method helpers.

12.1 Class method macros

The CLASS_BOOST module provides macros for extending class functionality, including the `[serialize_as_class]` annotation for automatic serialization and common class patterns like abstract method enforcement.

All functions and symbols are in "class_boost" module, use require to get access to it.

```
require daslib/class_boost
```

12.1.1 Function annotations

class_boost::**class_method**

Turns a static method into a class method by adding a `self` argument of the class type as the first argument, and wrapping the function body in `with (self) { ... }`. Applied via `[class_method]` annotation.

class_boost::**explicit_const_class_method**

Same as `[class_method]` but marks the `self` parameter with `explicitConst`, allowing overloading of const and non-const class methods.

12.2 instance_function function annotation

The INSTANCE_FUNCTION module provides the `[instance_function]` annotation for creating bound method-like functions. It captures the `self` reference at call time, enabling object-oriented dispatch patterns in daslang.

All functions and symbols are in "instance_function" module, use require to get access to it.

```
require daslib/instance_function
```

12.2.1 Function annotations

instance_function::**instance_function**

[instance_function(generic_name,type1=type1r,type2=type2r,…)] macro creates instance of the generic function with a particular set of types. In the followin example body of the function inst will be replaced with body of the function print_zero with type int:

```
def print_zero ( a : auto(TT) )
    print("{[[TT]]}\n")
[export, instance_function(print_zero,TT="int")]
def inst {}
```

12.3 Interfaces

The INTERFACES module implements interface-based polymorphism for daslang. It provides the `[interface]` annotation for defining abstract interfaces with virtual method tables, supporting multiple implementations and dynamic dispatch without class inheritance.

Features:

- Interface inheritance (`class IChild : IParent`)
- Default method implementations (non-abstract methods)
- Compile-time completeness checking (error 30111 on missing methods)
- `is`/`as`/`?as` operators via the `InterfaceAsIs` variant macro

All functions and symbols are in "interfaces" module, use require to get access to it.

```
require daslib/interfaces
```

Example:

```
require daslib/interfaces

[interface]
class IGreeter {
    def abstract greet(name : string) : string
}

[implements(IGreeter)]
class MyGreeter {
    def IGreeter`greet(name : string) : string {
        return "Hello, {name}!"
    }
}

[export]
def main() {
    var obj = new MyGreeter()
    var greeter = obj as IGreeter
    print("{greeter->greet("world")}\n")
```

(continues on next page)

(continued from previous page)

```
}
// output: Hello, world!
```

See also: Interfaces tutorial for a complete walkthrough.

12.3.1 Variant macros

`interfaces::`**`InterfaceAsIs`**

Variant macro that enables `is`, `as`, and `?as` operators for interface types declared with `[interface]` / `[implements]`.

- `ptr is IFoo` — compile-time check (`true` when the struct implements `IFoo`)
- `ptr as IFoo` — obtains the interface proxy via the generated getter
- `ptr ?as IFoo` — null-safe version: returns `null` when the pointer is null

12.3.2 Structure macros

`interfaces::`**`interface`**

Verifies that the annotated class is a valid interface — it may only contain function-typed fields (no data members). Applied via `[interface]` annotation.

`interfaces::`**`implements`**

Generates interface bindings for a struct. Creates a proxy class that delegates interface method calls to the struct's own methods, and adds a ``get`InterfaceName`` method that lazily constructs the proxy. Applied via `[implements(InterfaceName)]`.

If the interface inherits from a parent interface (`class IChild : IParent`), ancestor getters are generated automatically so that `is`/`as`/`?as` work with all ancestor interfaces.

At `finish` time, verifies that all abstract interface methods are implemented. Methods with default bodies in the interface class are optional — the proxy inherits them via class hierarchy.

12.4 Dynamic RTTI type casts

The DYNAMIC_CAST_RTTI module implements runtime dynamic casting between class types using RTTI information. It provides safe downcasting with null results on type mismatch, similar to C++ `dynamic_cast`.

All functions and symbols are in "dynamic_cast_rtti" module, use require to get access to it.

```
require daslib/dynamic_cast_rtti
```

12.4.1 Variant macros

`dynamic_cast_rtti::`**`ClassAsIs`**

Variant macro that implements class dynamic casting via *is* and *as*.

12.4.2 Dynamic casts

- *dynamic_type_cast (instance: auto; otherclass: auto(TT)) : TT?*
- *force_dynamic_type_cast (instance: auto; otherclass: auto(TT)) : TT?*
- *is_instance_of (instance: auto(TCLS)?; otherclass: auto(TT)) : auto*

`dynamic_cast_rtti::`**`dynamic_type_cast(instance: auto; otherclass: auto(TT)) : TT?`**`()`

Casts a class instance to the target type using RTTI, returns null if the cast fails.

Arguments

- **instance** : auto
- **otherclass** : auto(TT)

`dynamic_cast_rtti::`**`force_dynamic_type_cast(instance: auto; otherclass: auto(TT)) : TT?`**`()`

Casts a class instance to the target type using RTTI, panics if the cast fails.

Arguments

- **instance** : auto
- **otherclass** : auto(TT)

`dynamic_cast_rtti::`**`is_instance_of(instance: auto(TCLS)?; otherclass: auto(TT)) : auto`**`()`

Returns true if the class instance is an instance of the specified class using RTTI.

Arguments

- **instance** : auto(TCLS)?
- **otherclass** : auto(TT)

12.5 Generic return type instantiation

The GENERIC_RETURN module provides the `[generic_return]` annotation that allows generic functions to automatically deduce their return type from the body. This simplifies writing generic utility functions by eliminating explicit return type specifications.

All functions and symbols are in “generic_return” module, use require to get access to it.

```
require daslib/generic_return
```

12.5.1 Call macros

`generic_return::`**`generic_return`**

Replaces generic_return(expr) with a block that calls expr and returns its result, handling void, copyable, and movable return types.

CHAPTER

THIRTEEN

TESTING AND DEBUGGING

Code coverage, profiling, debug evaluation, test helpers (faker, fuzzer), and debug adapter protocol support.

13.1 Debug agent API

The DEBUGAPI module provides the debug agent infrastructure — creating, installing, and communicating with persistent debug agents that live in their own forked contexts. It supports cross-context function invocation, agent method calls, log interception, data and stack walking, instrumentation, and breakpoint management.

See tutorial_debug_agents and tutorial_data_walker for hands-on tutorials.

All functions and symbols are in “debugapi” module, use require to get access to it.

```
require debugapi
```

13.1.1 Structures

debugapi::**DapiArray**

Lightweight descriptor for an array encountered during data walking.

Fields

- **data** : void? - Pointer to the array data.
- **size** : uint - Number of elements currently in the array.
- **capacity** : uint - Allocated capacity of the array.
- **lock** : uint - Reference count lock for the array.
- **flags** : bitfield<_shared;_hopeless> - Bitfield flags (*_shared*, *_hopeless*).

debugapi::**DapiTable : DapiArray**

Lightweight descriptor for a table encountered during data walking. Extends *DapiArray*.

Fields

- **keys** : void? - Pointer to the table keys data.
- **hashes** : uint? - Pointer to the hash values array.

debugapi::**DapiBlock**

Lightweight descriptor for a block encountered during data walking.

Fields

- **stackOffset** : uint - Stack offset of the block.
- **argumentsOffset** : uint - Arguments offset on the stack.
- **body** : void? - Pointer to the block's simulation node (*SimNode*).
- **aotFunction** : void? - Pointer to the AOT-compiled function (if any).
- **functionArguments** : float4? - Pointer to the block's function arguments.
- **info** : *FuncInfo*? - Pointer to the block's *FuncInfo* metadata.

debugapi::**DapiFunc**

Lightweight descriptor for a function pointer encountered during data walking.

Fields

- **index** : int - Index of the function in the context's function table.

debugapi::**DapiLambda**

Lightweight descriptor for a lambda encountered during data walking.

Fields

- **captured** : void? - Pointer to the lambda's captured variable block.

debugapi::**DapiSequence**

Lightweight descriptor for an iterator/sequence encountered during data walking.

Fields

- **iter** : void? - Pointer to the underlying *Iterator* object.

13.1.2 Handled structures

debugapi::**Prologue**

Annotation for inspecting function call prologues. Provides access to call metadata inside the annotated block.

Fields

- **_block** : block<void>? - The block to execute with prologue context.
- **info** : *FuncInfo*? - Pointer to the *FuncInfo* of the called function.
- **arguments** : any? - Pointer to the function argument values.
- **fileName** : string - Source file name of the call site.
- **cmres** : void? - Pointer to the copy-on-return result slot.
- **line** : *LineInfo*? - Line number of the call site.
- **stackSize** : int - Stack size used by the function.

`debugapi::`**`DebugAgent`**

Fields

- **isThreadLocal** : bool - Annotation for declaring a debug agent class. Generates the boilerplate to subclass *DebugAgent* from C++. The optional *isThreadLocal* field installs as a thread-local agent.

`debugapi::`**`DataWalker`**

Annotation for declaring a data walker class. Generates the boilerplate to subclass *DataWalker* from C++.

`debugapi::`**`StackWalker`**

Annotation for declaring a stack walker class. Generates the boilerplate to subclass *StackWalker* from C++.

13.1.3 Classes

`debugapi::`**`DapiDebugAgent`**

Fields

- **thisAgent** : *DebugAgent*? - Base class for debug agents. Subclass and override methods (*onLog*, *onCollect*, *onBreakpoint*, etc.) to intercept runtime events. Register with *install_new_debug_agent*.

`debugapi::`**`DapiDataWalker`**

Base class for walking daslang data structures. Subclass and override per-type visitor methods (*Int*, *Float*, *String*, *Array*, *Table*, etc.) to inspect values. Create with *make_data_walker*.

`debugapi::`**`DapiStackWalker`**

Base class for walking the call stack. Subclass and override *onBeforeCall*, *onAfterPop*, *onArgument*, *onVariable*, etc. Create with *make_stack_walker*.

13.1.4 Agent lifecycle

- *delete_debug_agent_context (category: string)*
- *fork_debug_agent_context (function: function<():void>)*
- *get_debug_agent_context (category: string) : Context&*
- *has_debug_agent_context (category: string) : bool*
- *install_debug_agent (agent: smart_ptr<DebugAgent>; category: string)*
- *install_debug_agent_thread_local (agent: smart_ptr<DebugAgent>)*
- *install_new_debug_agent (var agentPtr: auto; category: string) : auto*
- *install_new_thread_local_debug_agent (var agentPtr: auto) : auto*
- *is_in_debug_agent_creation () : bool*
- *lock_debug_agent (block: block<():void>)*

debugapi::**delete_debug_agent_context**(*category: string*)

Removes the debug agent with the given category name. Notifies all other agents via *onUninstall*, then destroys the agent and its context. Safe no-op if the agent does not exist.

Arguments

- **category** : string implicit

debugapi::**fork_debug_agent_context**(*function: function<():void>*)

Clones the current context and calls the setup function inside the clone. The cloned context becomes an agent context that stays resident for the lifetime of the program.

Arguments

- **function** : function<void>

debugapi::**get_debug_agent_context(category: string) : Context&**()

Returns a reference to the *Context* of the named debug agent. Panics if the agent does not exist check with *has_debug_agent_context* first.

Arguments

- **category** : string implicit

debugapi::**has_debug_agent_context(category: string) : bool**()

Returns *true* if a debug agent with the given category name is currently installed.

Arguments

- **category** : string implicit

debugapi::**install_debug_agent**(*agent: smart_ptr<DebugAgent>; category: string*)

Installs a low-level *smart_ptr<DebugAgent>* under the given category name. Prefer *install_new_debug_agent* for the high-level pattern.

Arguments

- **agent** : smart_ptr< *DebugAgent*> implicit
- **category** : string implicit

debugapi::**install_debug_agent_thread_local**(*agent: smart_ptr<DebugAgent>*)

Installs a low-level *smart_ptr<DebugAgent>* as the thread-local debug agent.

Arguments

- **agent** : smart_ptr< *DebugAgent*> implicit

debugapi::**install_new_debug_agent(agentPtr: auto; category: string) : auto**()

Creates and installs a debug agent from a *DapiDebugAgent* subclass instance. This is the recommended high-level function for agent installation.

Arguments

- **agentPtr** : auto
- **category** : string

debugapi::**install_new_thread_local_debug_agent(agentPtr: auto) : auto**()

Creates and installs a thread-local debug agent from a *DapiDebugAgent* subclass instance.

Arguments

- **agentPtr** : auto

debugapi::**is_in_debug_agent_creation() : bool**()

Returns *true* if the current thread is inside a *fork_debug_agent_context* call. Used in auto-start guards to prevent recursive agent creation.

debugapi::**lock_debug_agent**(*block: block<():void>*)

Warning: This is unsafe operation.

Executes the block while holding the global debug agent mutex. Use when iterating or modifying agent state that must be thread-safe.

Arguments

- **block** : block<void> implicit

13.1.5 Cross-context invocation

- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string; arg2: any; arg3: any)*
- *invoke_debug_agent_function (arg0: string; arg1: string)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any)*

- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any)*
- *invoke_debug_agent_method (arg0: string; arg1: string)*
- *invoke_debug_agent_method (arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any)*

- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any)*
- *invoke_in_context (arg0: any; arg1: string)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any)*
- *invoke_in_context (arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any)*
- *invoke_in_context (arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any)*
- *invoke_in_context (arg0: any; arg1: lambda<():void>; arg2: any; arg3: any)*

invoke_debug_agent_function

`debugapi::`**`invoke_debug_agent_function`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any*)

Warning: This is unsafe operation.

Calls an *[export, pinvoke]* function in the named agent's context. Similar to *invoke_in_context* but resolves the agent context automatically from the category name.

Arguments

- **arg0** : string
- **arg1** : string
- **arg2** : any
- **arg3** : any
- **arg4** : any
- **arg5** : any
- **arg6** : any

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string; arg2: any; arg3: any*)

debugapi::**invoke_debug_agent_function**(*arg0: string; arg1: string*)

---

invoke_debug_agent_method

debugapi::**invoke_debug_agent_method**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any*)

Warning: This is unsafe operation.

Calls a method on the debug agent's class instance by name. The first argument is the agent category, the second is the method name, followed by up to 10 user arguments. The agent's *self* is passed automatically.

Arguments

- **arg0** : string
- **arg1** : string
- **arg2** : any
- **arg3** : any
- **arg4** : any
- **arg5** : any
- **arg6** : any
- **arg7** : any

- **arg8** : any
- **arg9** : any
- **arg10** : any
- **arg11** : any

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string*)

`debugapi::`**`invoke_debug_agent_method`**(*arg0: string; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any*)

---

invoke_in_context

`debugapi::`**`invoke_in_context`**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any*)

Warning: This is unsafe operation.

Calls a function in another context. Accepts a *Context* reference and either a function name (string), a *function* pointer, or a *lambda*, plus up to 10 extra arguments. Target functions must be marked *[export, pinvoke]*.

Arguments

- **arg0** : any
- **arg1** : function<void>
- **arg2** : any
- **arg3** : any
- **arg4** : any
- **arg5** : any

- **arg6** : any
- **arg7** : any
- **arg8** : any
- **arg9** : any
- **arg10** : any
- **arg11** : any

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any*) debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any; arg11: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any; arg10: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: string; arg2: any; arg3: any; arg4: any; arg5: any; arg6: any; arg7: any; arg8: any; arg9: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: function<():void>; arg2: any; arg3: any; arg4: any*)

debugapi::**invoke_in_context**(*arg0: any; arg1: lambda<():void>; arg2: any; arg3: any*)

13.1.6 Agent construction

- *make_data_walker (class: void?; info: StructInfo const?) : smart_ptr<DataWalker>*
- *make_data_walker (classPtr: auto) : smart_ptr<DataWalker>*
- *make_debug_agent (class: void?; info: StructInfo const?) : smart_ptr<DebugAgent>*
- *make_stack_walker (class: void?; info: StructInfo const?) : smart_ptr<StackWalker>*
- *make_stack_walker (classPtr: auto) : smart_ptr<StackWalker>*

make_data_walker

debugapi::**make_data_walker(class: void?; info: StructInfo const?) : smart_ptr<DataWalker>**()

Wraps a *DapiDataWalker* class pointer into a *smart_ptr<DataWalker>* for use with *walk_data*.

Arguments

- **class** : void? implicit
- **info** : *StructInfo*? implicit

debugapi::**make_data_walker(classPtr: auto) : smart_ptr<DataWalker>()**

---

debugapi::**make_debug_agent(class: void?; info: StructInfo const?) : smart_ptr<DebugAgent>()**

Low-level constructor that wraps a *DapiDebugAgent* class pointer into a *smart_ptr<DebugAgent>*. Called internally by *install_new_debug_agent*.

Arguments

- **class** : void? implicit
- **info** : *StructInfo*? implicit

make_stack_walker

debugapi::**make_stack_walker(class: void?; info: StructInfo const?) : smart_ptr<StackWalker>()**

Wraps a *DapiStackWalker* class pointer into a *smart_ptr<StackWalker>* for use with *walk_stack*.

Arguments

- **class** : void? implicit
- **info** : *StructInfo*? implicit

debugapi::**make_stack_walker(classPtr: auto) : smart_ptr<StackWalker>()**

13.1.7 Agent tick and state collection

- *collect_debug_agent_state (context: Context; at: LineInfo)*
- *debug_agent_command (command: string)*
- *debugger_stop_requested () : bool*
- *on_breakpoints_reset (file: string; breakpointsNum: int)*
- *report_context_state (context: Context; category: string; name: string; info: TypeInfo const?; data: void?)*
- *tick_debug_agent ()*
- *tick_debug_agent (agent: string)*

debugapi::**collect_debug_agent_state**(*context: Context; at: LineInfo*)

Triggers *onCollect* on all installed debug agents, passing the calling context and line info. Agents can inspect the context and report variables via *report_context_state*.

Arguments

- **context** : *Context* implicit
- **at** : *LineInfo* implicit

debugapi::**debug_agent_command**(*command: string*)

Sends a user-defined command string to all debug agents. Agents receive it in *onUserCommand*.

Arguments

- **command** : string implicit

`debugapi::`**`debugger_stop_requested() : bool`**`()`

Returns *true* if any debug agent has requested the program to stop (e.g. via a breakpoint or user interrupt).

`debugapi::`**`on_breakpoints_reset`**`(`*file: string; breakpointsNum: int*`)`

Notifies all debug agents that breakpoints for the given file have been reset. The *breakpointsNum* parameter indicates the new breakpoint count.

Arguments

- **file** : string implicit
- **breakpointsNum** : int

`debugapi::`**`report_context_state`**`(`*context: Context; category: string; name: string; info: TypeInfo const?; data: void?*`)`

Reports a named variable from inside *onCollect* back to the debug system. Each call triggers *onVariable* on all agents with the category, name, type info, and data pointer.

Arguments

- **context** : *Context* implicit
- **category** : string implicit
- **name** : string implicit
- **info** : *TypeInfo*? implicit
- **data** : void? implicit

tick_debug_agent

`debugapi::`**`tick_debug_agent`**`()`

Calls *onTick* on all installed debug agents (no-arg variant), or on a specific named agent (string variant). Typically called once per frame by the host application.

`debugapi::`**`tick_debug_agent`**`(`*agent: string*`)`

13.1.8 Instrumentation

- *clear_instruments (context: Context)*
- *instrument_all_functions (context: Context)*
- *instrument_all_functions (ctx: Context; block: block<(function<():void>;SimFunction const?):uint64>)*
- *instrument_all_functions_thread_local (ctx: Context; block: block<(function<():void>;SimFunction const?):uint64>)*
- *instrument_all_functions_thread_local (context: Context)*
- *instrument_context_allocations (context: Context; isInstrumenting: bool)*
- *instrument_function (context: Context; function: function<():void>; isInstrumenting: bool; userData: uint64)*
- *instrument_node (context: Context; isInstrumenting: bool; block: block<(LineInfo):bool>)*
- *set_single_step (context: Context; enabled: bool)*

`debugapi::`**`clear_instruments`**`(`*context: Context*`)`

Removes all instrumentation from the given context, restoring original execution.

Arguments

- **context** : *Context* implicit

instrument_all_functions

`debugapi::`**`instrument_all_functions`**`(`*context: Context*`)`

Enables or disables instrumentation for all functions in the given context. Overloaded: the no-block variant instruments everything; the block variant calls a filter to select functions.

Arguments

- **context** : *Context* implicit

`debugapi::`**`instrument_all_functions`**`(`*ctx: Context; block: block<(function<():void>;SimFunction const?):uint64>*`)`

---

instrument_all_functions_thread_local

`debugapi::`**`instrument_all_functions_thread_local`**`(`*ctx: Context; block: block<(function<():void>;SimFunction const?):uint64>*`)`

Thread-local variant of *instrument_all_functions*. Instruments functions only on the current thread.

Arguments

- **ctx** : *Context* implicit
- **block** : block<(function<void>; *SimFunction*?):uint64> implicit

`debugapi::`**`instrument_all_functions_thread_local`**`(`*context: Context*`)`

`debugapi::`**`instrument_context_allocations`**`(`*context: Context; isInstrumenting: bool*`)`

Enables or disables allocation tracking on the given context. When enabled, the debug agent receives *onAllocate* / *onReallocate* / *onFree* callbacks.

Arguments

- **context** : *Context* implicit
- **isInstrumenting** : bool

`debugapi::`**`instrument_function`**`(`*context: Context; function: function<():void>; isInstrumenting: bool; userData: uint64*`)`

Enables or disables instrumentation for a specific function in the given context. The *userData* value is passed to the instrumentation callback.

Arguments

- **context** : *Context* implicit
- **function** : function<void>
- **isInstrumenting** : bool
- **userData** : uint64

`debugapi::`**`instrument_node`**(*context: Context; isInstrumenting: bool; block: block<(LineInfo):bool>*)

Enables or disables per-node instrumentation on the given context. The block receives each *LineInfo* and returns *true* to instrument that node.

Arguments

- **context** : *Context* implicit
- **isInstrumenting** : bool
- **block** : block<(*LineInfo*):bool> implicit

`debugapi::`**`set_single_step`**(*context: Context; enabled: bool*)

Enables or disables single-step execution on the given context. When enabled, the debug agent receives *onSingleStep* for each statement.

Arguments

- **context** : *Context* implicit
- **enabled** : bool

13.1.9 Data and stack walking

- *stack_depth (context: Context) : int*
- *stackwalk (context: Context; line: LineInfo)*
- *walk_data (walker: smart_ptr<DataWalker>; data: void?; struct_info: StructInfo)*
- *walk_data (walker: smart_ptr<DataWalker>; data: void?; info: TypeInfo)*
- *walk_data (walker: smart_ptr<DataWalker>; data: float4; info: TypeInfo)*
- *walk_stack (walker: smart_ptr<StackWalker>; context: Context; line: LineInfo)*

`debugapi::`**`stack_depth(context: Context) : int`**()

Returns the current call stack depth of the given context.

Arguments

- **context** : *Context* implicit

`debugapi::`**`stackwalk`**(*context: Context; line: LineInfo*)

Prints a human-readable stack trace of the given context to the log output.

Arguments

- **context** : *Context* implicit
- **line** : *LineInfo* implicit

walk_data

debugapi::**walk_data**(*walker: smart_ptr<DataWalker>; data: void?; struct_info: StructInfo*)

Walks a daslang data structure using the provided *DataWalker*. The walker receives typed callbacks for each value encountered. Overloaded for raw data+`StructInfo`, *float4`+`TypeInfo*, and *void?`+`TypeInfo*.

Arguments

- **walker** : smart_ptr< *DataWalker*> implicit
- **data** : void? implicit
- **struct_info** : *StructInfo* implicit

debugapi::**walk_data**(*walker: smart_ptr<DataWalker>; data: void?; info: TypeInfo*)

debugapi::**walk_data**(*walker: smart_ptr<DataWalker>; data: float4; info: TypeInfo*)

---

debugapi::**walk_stack**(*walker: smart_ptr<StackWalker>; context: Context; line: LineInfo*)

Walks the call stack of the given context using the provided *StackWalker*. The walker receives callbacks for each stack frame, argument, and local variable.

Arguments

- **walker** : smart_ptr< *StackWalker*> implicit
- **context** : *Context* implicit
- **line** : *LineInfo* implicit

13.1.10 Context inspection

- *get_context_global_variable (context: any; index: int) : void?*
- *get_context_global_variable (context: any; name: string) : void?*
- *get_heap_stats (context: Context; bytes: uint64?)*
- *has_function (context: Context; function_name: string) : bool*

get_context_global_variable

debugapi::**get_context_global_variable(context: any; index: int) : void?**()

Warning: This is unsafe operation.

Returns a pointer to a global variable in the given context, looked up by name (string variant) or by index (int variant). Returns *null* if not found.

Arguments

- **context** : any
- **index** : int

debugapi::**get_context_global_variable(context: any; name: string) : void?**()

---

debugapi::**get_heap_stats**(*context: Context; bytes: uint64?*)

> **Warning:** This is unsafe operation.

Writes the heap allocation statistics of the given context into the provided *uint64* pointer.

Arguments

- **context** : *Context* implicit
- **bytes** : uint64? implicit

debugapi::**has_function(context: Context; function_name: string) : bool**()

Returns *true* if the given context contains an exported function with the specified name.

Arguments

- **context** : *Context* implicit
- **function_name** : string implicit

13.1.11 Breakpoints

- *clear_hw_breakpoint (arg0: int) : bool*
- *set_hw_breakpoint (context: Context; address: void?; size: int; writeOnly: bool) : int*

debugapi::**clear_hw_breakpoint(arg0: int) : bool**()

> **Warning:** This is unsafe operation.

Clears the hardware breakpoint with the given index. Returns *true* on success.

Arguments

- **arg0** : int

debugapi::**set_hw_breakpoint(context: Context; address: void?; size: int; writeOnly: bool) : int**()

> **Warning:** This is unsafe operation.

Sets a hardware breakpoint at the given memory address. Returns the breakpoint index, or -1 on failure. The *writeOnly* flag selects write-only vs. read/write watch.

Arguments

- **context** : *Context* implicit
- **address** : void? implicit
- **size** : int
- **writeOnly** : bool

13.1.12 Memory

- *break_on_free (context: Context; ptr: void?; size: uint)*
- *free_temp_string (context: Context)*
- *temp_string_size (context: Context) : uint64*
- *track_insane_pointer (ptr: void?)*

debugapi::**break_on_free**(*context: Context; ptr: void?; size: uint*)

Warning: This is unsafe operation.

Triggers a debug break when the specified memory region is freed. Useful for tracking down use-after-free bugs.

Arguments

- **context** : *Context* implicit
- **ptr** : void? implicit
- **size** : uint

debugapi::**free_temp_string**(*context: Context*)

Warning: This is unsafe operation.

Frees all temporary string allocations in the given context.

Arguments

- **context** : *Context* implicit

debugapi::**temp_string_size(context: Context) : uint64**()

Warning: This is unsafe operation.

Returns the total size in bytes of temporary string allocations in the given context.

Arguments

- **context** : *Context* implicit

debugapi::**track_insane_pointer**(*ptr: void?*)

Warning: This is unsafe operation.

Begins tracking the specified pointer for dangling-reference detection.

Arguments

- **ptr** : void? implicit

13.2 Code coverage profiling

The COVERAGE module provides code coverage instrumentation. It automatically inserts coverage tracking calls at every block entry point, allowing you to measure which parts of your code are executed during a run.

All functions and symbols are in “coverage” module, use require to get access to it.

```
require daslib/coverage
```

Note: This module uses `[infer_macro]` to instrument code at compile time. Coverage results can be collected and analyzed after program execution.

13.3 Instrumenting profiler

The PROFILER module provides CPU profiling infrastructure for measuring function execution times. It includes instrumentation-based profiling with hierarchical call tracking and timing statistics.

See also *Profiler cross-context helpers* for cross-context profiler control and scoped timing macros.

All functions and symbols are in “profiler” module, use require to get access to it.

```
require daslib/profiler
```

13.3.1 Profiler control

- *set_enable_profiler (var ctxId: uint64; enabled: bool)*

`profiler::`**`set_enable_profiler`**`(`*ctxId: uint64; enabled: bool*`)`

Enables or disables the profiler for the given context ID.

Arguments

- **ctxId** : uint64
- **enabled** : bool

13.4 Profiler cross-context helpers

The PROFILER_BOOST module extends profiling with high-level macros for scoped timing (`profile_block`), function-level profiling annotations, and formatted output of profiling results.

See also *Instrumenting profiler* for the core profiling infrastructure.

All functions and symbols are in “profiler_boost” module, use require to get access to it.

```
require daslib/profiler_boost
```

13.4.1 Context profiler control

- *disable_profiler (var ctx: Context)*
- *enable_profiler (var ctx: Context)*

`profiler_boost::`**`disable_profiler`**`(`*ctx: Context*`)`

Disables the profiler for the given context.

Arguments

- **ctx** : *Context*

`profiler_boost::`**`enable_profiler`**`(`*ctx: Context*`)`

Enables the profiler for the given context.

Arguments

- **ctx** : *Context*

13.5 Debug expression evaluator

The DEBUG_EVAL module provides runtime expression evaluation for debugging purposes. It can evaluate daslang expressions in the context of a running program, supporting variable inspection and interactive debugging.

All functions and symbols are in "debug_eval" module, use require to get access to it.

```
require daslib/debug_eval
```

13.5.1 Structures

`debug_eval::`**`Result`**

Result of evaluating a debug expression.

Fields

- **tinfo** : *TypeInfo* - Type information of the result.
- **value** : float4 - Raw value storage for the result.
- **data** : void? - Pointer to the result data, if available.
- **error** : string - Error message, empty if evaluation succeeded.

13.5.2 Evaluation

- *debug_eval (context: table<string, Result>; expr: string) : Result*

`debug_eval::`**`debug_eval(context: table<string, Result>; expr: string) : Result`**`()`

Evaluates a debug expression string with the given variable context and returns the result.

Arguments

- **context** : table<string; *Result*>
- **expr** : string

13.6 Faker

Random test-data generator.

The `Faker` struct produces random values for every built-in type (integers, floats, vectors, strings, dates, booleans) using configurable ranges. Used by `fuzzer` for fuzz testing.

All functions and symbols are in “faker” module, use require to get access to it.

```
require daslib/faker
```

13.6.1 Structures

faker::**Faker**

Instance of the faker with all the settings inside.

Fields

- **min_year** : uint = 0x7bc - minimal faker’s year
- **total_years** : uint = 0x2a - faker year’s range
- **rnd** : iterator<uint> = each_random_uint(13) - fakers random number generator
- **max_long_string** : uint = 0x1000 - maximal length of generated string

13.6.2 Constructor

- *Faker (var rng: iterator<uint>) : Faker*

faker::**Faker(rng: iterator<uint>) : Faker**()

Constructs a Faker instance with the given random number generator.

Arguments

- **rng** : iterator<uint>

13.6.3 Random values

- *random_double (var faker: Faker) : double*
- *random_float (var faker: Faker) : float*
- *random_float2 (var faker: Faker) : float2*
- *random_float3 (var faker: Faker) : float3*
- *random_float3x3 (var faker: Faker) : float3x3*
- *random_float3x4 (var faker: Faker) : float3x4*
- *random_float4 (var faker: Faker) : float4*
- *random_float4x4 (var faker: Faker) : float4x4*
- *random_int (var faker: Faker) : int*
- *random_int16 (var faker: Faker) : int16*

- *random_int2 (var faker: Faker) : int2*
- *random_int3 (var faker: Faker) : int3*
- *random_int4 (var faker: Faker) : int4*
- *random_int64 (var faker: Faker) : int64*
- *random_int8 (var faker: Faker) : int8*
- *random_range (var faker: Faker) : range*
- *random_range64 (var faker: Faker) : range64*
- *random_uint (var faker: Faker) : uint*
- *random_uint16 (var faker: Faker) : uint16*
- *random_uint2 (var faker: Faker) : uint2*
- *random_uint3 (var faker: Faker) : uint3*
- *random_uint4 (var faker: Faker) : uint4*
- *random_uint64 (var faker: Faker) : uint64*
- *random_uint8 (var faker: Faker) : uint8*
- *random_urange (var faker: Faker) : urange*
- *random_urange64 (var faker: Faker) : urange64*

faker::**random_double(faker: Faker) : double**()

Generates random double.

Arguments

- **faker** : *Faker*

faker::**random_float(faker: Faker) : float**()

Generates random float.

Arguments

- **faker** : *Faker*

faker::**random_float2(faker: Faker) : float2**()

Generates random float2.

Arguments

- **faker** : *Faker*

faker::**random_float3(faker: Faker) : float3**()

Generates random float3.

Arguments

- **faker** : *Faker*

faker::**random_float3x3(faker: Faker) : float3x3**()

Generates random float3x3.

Arguments

- **faker** : *Faker*

`faker::random_float3x4(faker: Faker) : float3x4()`

Generates random float3x4.

Arguments

- **faker** : *Faker*

`faker::random_float4(faker: Faker) : float4()`

Generates random float4.

Arguments

- **faker** : *Faker*

`faker::random_float4x4(faker: Faker) : float4x4()`

Generates random float4x4.

Arguments

- **faker** : *Faker*

`faker::random_int(faker: Faker) : int()`

Generates random integer.

Arguments

- **faker** : *Faker*

`faker::random_int16(faker: Faker) : int16()`

Generates random int16.

Arguments

- **faker** : *Faker*

`faker::random_int2(faker: Faker) : int2()`

Generates random int2.

Arguments

- **faker** : *Faker*

`faker::random_int3(faker: Faker) : int3()`

Generates random int3.

Arguments

- **faker** : *Faker*

`faker::random_int4(faker: Faker) : int4()`

Generates random int4.

Arguments

- **faker** : *Faker*

`faker::`**`random_int64(faker: Faker) : int64`**`()`

Generates random int64

Arguments

- **faker** : *Faker*

`faker::`**`random_int8(faker: Faker) : int8`**`()`

Generates random int8.

Arguments

- **faker** : *Faker*

`faker::`**`random_range(faker: Faker) : range`**`()`

Generates random range.

Arguments

- **faker** : *Faker*

`faker::`**`random_range64(faker: Faker) : range64`**`()`

Generates random range64.

Arguments

- **faker** : *Faker*

`faker::`**`random_uint(faker: Faker) : uint`**`()`

Generates random unsigned integer.

Arguments

- **faker** : *Faker*

`faker::`**`random_uint16(faker: Faker) : uint16`**`()`

Generates random uint16.

Arguments

- **faker** : *Faker*

`faker::`**`random_uint2(faker: Faker) : uint2`**`()`

Generates random uint2.

Arguments

- **faker** : *Faker*

`faker::`**`random_uint3(faker: Faker) : uint3`**`()`

Generates random uint3.

Arguments

- **faker** : *Faker*

faker::**random_uint4(faker: Faker) : uint4**()

Generates random uint4.

Arguments

- **faker** : *Faker*

faker::**random_uint64(faker: Faker) : uint64**()

Generates random uint64

Arguments

- **faker** : *Faker*

faker::**random_uint8(faker: Faker) : uint8**()

Generates random uint8.

Arguments

- **faker** : *Faker*

faker::**random_urange(faker: Faker) : urange**()

Generates random urange.

Arguments

- **faker** : *Faker*

faker::**random_urange64(faker: Faker) : urange64**()

Generates random urange64.

Arguments

- **faker** : *Faker*

13.6.4 Random strings

- *any_char (var faker: Faker) : int*
- *any_enum (var faker: Faker; enum_value: auto(TT)) : TT*
- *any_file_name (var faker: Faker) : string*
- *any_float (var faker: Faker) : string*
- *any_hex (var faker: Faker) : string*
- *any_int (var faker: Faker) : string*
- *any_set (var faker: Faker) : uint[8]*
- *any_string (var faker: Faker) : string*
- *any_uint (var faker: Faker) : string*
- *long_string (var faker: Faker) : string*
- *number (var faker: Faker) : string*
- *positive_int (var faker: Faker) : string*

faker::**any_char(faker: Faker) : int**()

Generates random char. (1 to 255 range)

Arguments

- **faker** : *Faker*

faker::**any_enum(faker: Faker; enum_value: auto(TT)) : TT**()

Generates random enumeration value.

Arguments

- **faker** : *Faker*
- **enum_value** : auto(TT)

faker::**any_file_name(faker: Faker) : string**()

Generates random file name.

Arguments

- **faker** : *Faker*

faker::**any_float(faker: Faker) : string**()

Generates random float string.

Arguments

- **faker** : *Faker*

faker::**any_hex(faker: Faker) : string**()

Generates random integer hex string.

Arguments

- **faker** : *Faker*

faker::**any_int(faker: Faker) : string**()

Generates random integer string.

Arguments

- **faker** : *Faker*

faker::**any_set(faker: Faker) : uint[8]**()

Generates random set (uint[8])

Arguments

- **faker** : *Faker*

faker::**any_string(faker: Faker) : string**()

Generates a string of random characters. The string is anywhere between 0 and regex::re_gen_get_rep_limit() characters long.

Arguments

- **faker** : *Faker*

`faker::`**`any_uint(faker: Faker) : string`**`()`

Generates random unsigned integer string.

Arguments

- **faker** : *Faker*

`faker::`**`long_string(faker: Faker) : string`**`()`

Generates a long string of random characters. The string is anywhere between 0 and faker.max_long_string characters long.

Arguments

- **faker** : *Faker*

`faker::`**`number(faker: Faker) : string`**`()`

Generates random number string.

Arguments

- **faker** : *Faker*

`faker::`**`positive_int(faker: Faker) : string`**`()`

Generates random positive integer string.

Arguments

- **faker** : *Faker*

13.6.5 Date and time

- *date (var faker: Faker) : string*
- *day (var faker: Faker) : string*
- *is_leap_year (year: uint) : bool*
- *month (var faker: Faker) : string*
- *week_day (year: uint; month: uint; day: uint) : int*
- *week_day (year: int; month: int; day: int) : int*

`faker::`**`date(faker: Faker) : string`**`()`

Generates random date string.

Arguments

- **faker** : *Faker*

`faker::`**`day(faker: Faker) : string`**`()`

Generates random day string.

Arguments

- **faker** : *Faker*

faker::**is_leap_year(year: uint) : bool**()

Returns true if year is leap year.

Arguments

- **year** : uint

faker::**month(faker: Faker) : string**()

Generates random month string.

Arguments

- **faker** : *Faker*

week_day

faker::**week_day(year: uint; month: uint; day: uint) : int**()

Returns week day for given date.

Arguments

- **year** : uint
- **month** : uint
- **day** : uint

faker::**week_day(year: int; month: int; day: int) : int**()

13.7 Fuzzer

The FUZZER module implements fuzz testing infrastructure for daslang programs. It generates random inputs for functions and verifies they do not crash or produce unexpected errors, helping discover edge cases and robustness issues.

All functions and symbols are in “fuzzer” module, use require to get access to it.

```
require daslib/fuzzer
```

13.7.1 Fuzzer tests

- *fuzz (fuzz_count: int; blk: block<():void>)*
- *fuzz (blk: block<():void>)*
- *fuzz_all_ints_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_all_unsigned_ints_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_compareable_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_debug (fuzz_count: int; blk: block<():void>)*
- *fuzz_debug (blk: block<():void>)*
- *fuzz_eq_neq_op2 (t: auto; var fake: Faker; funcname: string) : auto*

- *fuzz_float_double_or_float_vec_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_float_double_or_float_vec_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_float_double_or_float_vec_op3 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_float_or_float_vec_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_float_or_float_vec_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_int_vector_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_and_storage_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_and_vector_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_and_vector_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_and_vector_op2_no_unint_vec (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_and_vector_signed_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_op1 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_op3 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_op4 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_scal_vec_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_numeric_vec_scal_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_rotate_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_shift_op2 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_vec_mad_op3 (t: auto; var fake: Faker; funcname: string) : auto*
- *fuzz_vec_op3 (t: auto; var fake: Faker; funcname: string) : auto*

fuzz

`fuzzer::`**`fuzz`**(*fuzz_count: int; blk: block<():void>*)

run block however many times ignore panic, so that we can see that runtime crashes

Arguments

- **fuzz_count** : int
- **blk** : block<void>

`fuzzer::`**`fuzz`**(*blk: block<():void>*)

`fuzzer::`**`fuzz_all_ints_op1(t: auto; fake: Faker; funcname: string) : auto`**`()`

fuzzes generic function that takes single numeric or vector argument. arguments are: int, uint, int64, uint64

Arguments

- **t** : auto
- **fake** : *Faker*

- **funcname** : string

fuzzer::**fuzz_all_unsigned_ints_op1(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes single numeric or vector argument. arguments are: uint, uint64

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_compareable_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, float, double, int64, uint64, string

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzz_debug

fuzzer::**fuzz_debug**(*fuzz_count: int; blk: block<():void>*)

run block however many times do not ignore panic, so that we can see where the runtime fails this is here so that *fuzz* can be easily replaced with *fuzz_debug* for the purpose of debugging

Arguments

- **fuzz_count** : int
- **blk** : block<void>

fuzzer::**fuzz_debug**(*blk: block<():void>*)

---

fuzzer::**fuzz_eq_neq_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, int64, uint64, float, double, string, int2, int3, int4, uint2, uint3, uint4, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_float_double_or_float_vec_op1(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes single numeric or vector argument. arguments are: float, double, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*

- **funcname** : string

fuzzer::**fuzz_float_double_or_float_vec_op2(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes two numeric or vector arguments. arguments are: float, double, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_float_double_or_float_vec_op3(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes three numeric or vector arguments. arguments are: float, double, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_float_or_float_vec_op1(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes single numeric or vector argument. arguments are: float, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_float_or_float_vec_op2(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes two numeric or vector arguments. arguments are: float, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_int_vector_op2(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, int2, int3, int4, uint2, uint3, uint4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_and_storage_op1(t: auto; fake: Faker; funcname: string) : auto()**

fuzzes generic function that takes single numeric or vector argument. arguments are: int, uint, int8, uint8, int16, uint16, int64, uint64, float, double

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_and_vector_op1(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes single numeric or vector argument. arguments are: int, uint, float, double, string, int2, int3, int4, uint2, uint3, uint4, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_and_vector_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, float, double, int2, int3, int4, uint2, uint3, uint4, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_and_vector_op2_no_unint_vec(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, float, double, int2, int3, int4, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_and_vector_signed_op1(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes single numeric or vector argument. arguments are: int, uint, float, double, string, int2, int3, int4, uint2, uint3, uint4, float2, float3, float4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_op1(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes single numeric or vector argument. arguments are: int, uint, float, double

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes two numeric or vector arguments. arguments are: int, uint, float, double

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_op3(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes three numeric or vector arguments. arguments are: int, uint, float, double

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_op4(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes four numeric or vector arguments. arguments are: int, uint, float, double

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_scal_vec_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes vector and matching scalar on the left arguments pairs are: int2,int; int3,int; uint2,uint; uint3,uint; uint4,uint; int4,int; float2,float; float3,float; float4,float

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_numeric_vec_scal_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes vector and matching scalar on the right arguments pairs are: int2,int; int3,int; uint2,uint; uint3,uint; uint4,uint; int4,int; float2,float; float3,float; float4,float

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

fuzzer::**fuzz_rotate_op2(t: auto; fake: Faker; funcname: string) : auto**()

fuzzes generic function that takes numeric or vector argument, with matching rotate type on the right. arguments are: int, uint

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

`fuzzer::fuzz_shift_op2(t: auto; fake: Faker; funcname: string) : auto()`

fuzzes generic function that takes numeric or vector argument, with matching shift type on the right. arguments are: int, uint, int2, int3, int4, uint2, uint3, uint4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

`fuzzer::fuzz_vec_mad_op3(t: auto; fake: Faker; funcname: string) : auto()`

fuzzes generic function that takes three numeric or vector arguments. arguments are: float2, float3, float4, int2, int3, int4, uint2, uint3, uint4 second argument is float, int, uint accordingly

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

`fuzzer::fuzz_vec_op3(t: auto; fake: Faker; funcname: string) : auto()`

fuzzes generic function that takes three numeric or vector arguments. arguments are: float2, float3, float4, int2, int3, int4, uint2, uint3, uint4

Arguments

- **t** : auto
- **fake** : *Faker*
- **funcname** : string

13.8 Debug Adapter Protocol data structures

The DAP module implements the Debug Adapter Protocol (DAP) for integrating daslang with external debuggers. It provides the message types, serialization, and communication infrastructure needed for IDE debugging support.

All functions and symbols are in “dap” module, use require to get access to it.

```
require daslib/dap
```

13.8.1 Structures

`dap::InitializeRequestArguments`

Arguments for the DAP initialize request.

`dap::DisconnectArguments`

Arguments for the DAP disconnect request.

Fields

- **restart** : bool - Whether to restart the debuggee after disconnecting.
- **terminateDebuggee** : bool - Whether to terminate the debuggee when disconnecting.
- **suspendDebuggee** : bool - Whether to suspend the debuggee when disconnecting.

`dap::Capabilities`

Debugger capabilities reported in the initialize response.

Fields

- **supportsConfigurationDoneRequest** : bool - Whether the adapter supports the configuration-Done request.
- **supportsRestartRequest** : bool - Whether the adapter supports the restart request.
- **supportTerminateDebuggee** : bool - Whether the adapter supports terminating the debuggee.
- **supportsTerminateRequest** : bool - Whether the adapter supports the terminate request.
- **supportsExceptionOptions** : bool - Whether the adapter supports exception options.
- **supportsExceptionFilterOptions** : bool - Whether the adapter supports exception filter options.
- **supportsDelayedStackTraceLoading** : bool - Whether the adapter supports delayed stack trace loading.
- **supportsDataBreakpoints** : bool - Whether the adapter supports data breakpoints.

`dap::DataBreakpoint`

A data breakpoint that triggers on memory access.

Fields

- **dataId** : string - Identifier for the data to watch.
- **accessType** : string - Access type that triggers the breakpoint: read, write, or readWrite.
- **condition** : string - Optional expression condition for the breakpoint.
- **hitCondition** : string - Optional hit count condition for the breakpoint.
- **description** : string - Human-readable description of the breakpoint.
- **enabled** : bool - Whether the breakpoint is enabled.

`dap::SetDataBreakpointsArguments`

Arguments for the setDataBreakpoints request.

Fields

- **breakpoints** : array< *DataBreakpoint*> - Array of data breakpoints to set.

dap::`DataBreakpointInfoArguments`

Arguments for the dataBreakpointInfo request.

Fields

- **variablesReference** : double - Reference to the variable container.
- **name** : string - Name of the variable.

dap::`DataBreakpointInfoResponse`

Response body for the dataBreakpointInfo request.

Fields

- **dataId** : string - Identifier for the data, used when setting a data breakpoint.
- **description** : string - Human-readable description of the data.

dap::`SourceBreakpoint`

A breakpoint specified by source location line number.

Fields

- **line** : double - Line number of the breakpoint.

dap::`Source`

A source file descriptor with name and path.

Fields

- **name** : string - Short name of the source.
- **path** : string - Full file-system path of the source.

dap::`SetBreakpointsArguments`

Arguments for the setBreakpoints request.

Fields

- **source** : *Source* - Source file for which breakpoints are set.
- **breakpoints** : array< *SourceBreakpoint*> - Array of source breakpoints to set.
- **sourceModified** : bool - Whether the source has been modified since last build.

dap::`Breakpoint`

A breakpoint with verification status and location.

Fields

- **id** : double - Unique identifier for the breakpoint.
- **verified** : bool - Whether the breakpoint has been verified by the debugger.
- **source** : *Source* - Source file containing the breakpoint.
- **line** : double - Actual line number of the breakpoint.
- **message** : string - Optional message about the breakpoint state.

dap::**SetBreakpointsResponse**

Response body for the setBreakpoints request.

Fields

- **breakpoints** : array< *Breakpoint*> - Array of breakpoints with their verification status.

dap::**Thread**

A thread with an identifier and name.

Fields

- **id** : double - Unique identifier for the thread.
- **name** : string - Human-readable name of the thread.

dap::**ThreadsResponseBody**

Response body for the threads request.

Fields

- **threads** : array< *Thread*> - Array of threads.

dap::**StackTraceArguments**

Arguments for the stackTrace request.

Fields

- **threadId** : double - Thread for which to retrieve the stack trace.
- **startFrame** : double - Index of the first frame to return.
- **levels** : double - Maximum number of frames to return.

dap::**StackFrame**

A stack frame with source location and identifier.

Fields

- **id** : double - Unique identifier for the stack frame.
- **name** : string - Name of the frame, typically the function name.
- **source** : *Source* - Source file of the frame.
- **line** : double - Line number in the source file.
- **column** : double - Column number in the source file.

dap::**StackTraceResponseBody**

Response body for the stackTrace request.

Fields

- **stackFrames** : array< *StackFrame*> - Array of stack frames.
- **totalFrames** : double - Total number of frames available.

dap::**ScopesArguments**

Arguments for the scopes request.

Fields

- **frameId** : double - Stack frame for which to retrieve scopes.

dap::**Scope**

A named variable scope with a variables reference.

Fields

- **name** : string - Name of the scope (e.g. Locals, Arguments).
- **variablesReference** : double - Reference used to retrieve the variables of this scope.

dap::**ScopesResponseBody**

Response body for the scopes request.

Fields

- **scopes** : array< *Scope*> - Array of scopes for the given frame.

dap::**VariablesArguments**

Arguments for the variables request.

Fields

- **variablesReference** : double - Reference to the variable container to expand.
- **start** : double - Start index of variables to return (for paging).
- **count** : double - Number of variables to return (for paging).

dap::**Variable**

A variable with name, value, and type information.

Fields

- **name** : string - Name of the variable.
- **value** : string - String representation of the variable's value.
- **variablesReference** : double - Reference to child variables, if any.
- **_type** : string - Type of the variable.
- **indexedVariables** : double - Number of indexed child variables.

dap::**VariablesResponseBody**

Response body for the variables request.

Fields

- **variables** : array< *Variable*> - Array of variables.

dap::**OutputEventBody**

Body of the output event for debugger console messages.

Fields

- **category** : string - Category of the output (e.g. console, stdout, stderr).
- **output** : string - The output text.

dap::**ContinueArguments**

Arguments for the continue request.

Fields

- **threadId** : double - Thread to continue.

dap::**PauseArguments**

Arguments for the pause request.

Fields

- **threadId** : double - Thread to pause.

dap::**StepInArguments**

Arguments for the stepIn request.

Fields

- **threadId** : double - Thread to step into.

dap::**NextArguments**

Arguments for the next (step over) request.

Fields

- **threadId** : double - Thread to step over.

dap::**StepOutArguments**

Arguments for the stepOut request.

Fields

- **threadId** : double - Thread to step out of.

dap::**EvaluateArguments**

Arguments for the evaluate request.

Fields

- **expression** : string - Expression to evaluate.
- **frameId** : double - Stack frame in which to evaluate the expression.
- **context** : string - Context in which the expression is evaluated (e.g. watch, repl, hover).

dap::**EvaluateResponse**

Response body for the evaluate request.

Fields

- **result** : string - Type of the evaluation result.
- **_type** : string - String representation of the evaluation result.
- **variablesReference** : double - Reference to child variables of the result, if any.
- **indexedVariables** : double - Number of indexed child variables in the result.

`dap::`**`BreakpointEvent`**

Event body indicating a breakpoint status change.

Fields

- **reason** : string - Reason for the event: changed, new, or removed.
- **breakpoint** : *Breakpoint* - The breakpoint whose status changed.

`dap::`**`ThreadEvent`**

Event body indicating a thread started or exited.

Fields

- **reason** : string - Reason for the event: started or exited.
- **threadId** : double - Thread identifier.

13.8.2 JSON deserialization

- *ContinueArguments (data: JsonValue?) : ContinueArguments*
- *DataBreakpoint (data: JsonValue?) : DataBreakpoint*
- *DataBreakpointInfoArguments (data: JsonValue?) : DataBreakpointInfoArguments*
- *DisconnectArguments (data: JsonValue?) : DisconnectArguments*
- *EvaluateArguments (data: JsonValue?) : EvaluateArguments*
- *InitializeRequestArguments (data: JsonValue?) : InitializeRequestArguments*
- *NextArguments (data: JsonValue?) : NextArguments*
- *PauseArguments (data: JsonValue?) : PauseArguments*
- *ScopesArguments (data: JsonValue?) : ScopesArguments*
- *SetBreakpointsArguments (data: JsonValue?) : SetBreakpointsArguments*
- *SetDataBreakpointsArguments (data: JsonValue?) : SetDataBreakpointsArguments*
- *Source (data: JsonValue?) : Source*
- *SourceBreakpoint (data: JsonValue?) : SourceBreakpoint*
- *StackTraceArguments (data: JsonValue?) : StackTraceArguments*
- *StepInArguments (data: JsonValue?) : StepInArguments*
- *StepOutArguments (data: JsonValue?) : StepOutArguments*
- *VariablesArguments (data: JsonValue?) : VariablesArguments*

`dap::`**`ContinueArguments(data: JsonValue?) : ContinueArguments`**`()`

Constructs a ContinueArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**DataBreakpoint(data: JsonValue?) : DataBreakpoint**()

Constructs a DataBreakpoint from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**DataBreakpointInfoArguments(data: JsonValue?) : DataBreakpointInfoArguments**()

Constructs a DataBreakpointInfoArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**DisconnectArguments(data: JsonValue?) : DisconnectArguments**()

Constructs a DisconnectArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**EvaluateArguments(data: JsonValue?) : EvaluateArguments**()

Constructs an EvaluateArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**InitializeRequestArguments(data: JsonValue?) : InitializeRequestArguments**()

Constructs an InitializeRequestArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**NextArguments(data: JsonValue?) : NextArguments**()

Constructs a NextArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**PauseArguments(data: JsonValue?) : PauseArguments**()

Constructs a PauseArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**ScopesArguments(data: JsonValue?) : ScopesArguments**()

Constructs a ScopesArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**SetBreakpointsArguments(data: JsonValue?) : SetBreakpointsArguments**()

Constructs a SetBreakpointsArguments from a JSON value, parsing source and breakpoints.

Arguments

- **data** : *JsonValue*?

dap::**SetDataBreakpointsArguments(data: JsonValue?) : SetDataBreakpointsArguments**()

Constructs a SetDataBreakpointsArguments from a JSON value, parsing the breakpoints array.

Arguments

- **data** : *JsonValue*?

dap::**Source(data: JsonValue?) : Source**()

Constructs a Source from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**SourceBreakpoint(data: JsonValue?) : SourceBreakpoint**()

Constructs a SourceBreakpoint from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**StackTraceArguments(data: JsonValue?) : StackTraceArguments**()

Constructs a StackTraceArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**StepInArguments(data: JsonValue?) : StepInArguments**()

Constructs a StepInArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**StepOutArguments(data: JsonValue?) : StepOutArguments**()

Constructs a StepOutArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

dap::**VariablesArguments(data: JsonValue?) : VariablesArguments**()

Constructs a VariablesArguments from a JSON value.

Arguments

- **data** : *JsonValue*?

13.8.3 JSON serialization

- *JV (data: EvaluateResponse) : JsonValue?*
- *JV (data: Variable) : JsonValue?*

JV

`dap::JV(data: EvaluateResponse) : JsonValue?()`

Converts an EvaluateResponse struct to its DAP JSON representation.

Arguments

- **data** : *EvaluateResponse*

`dap::JV(data: Variable) : JsonValue?()`

13.8.4 JSON field accessors

- *j_s (val: JsonValue?; defVal: string = “”) : string*
- *job (val: JsonValue?; id: string; defVal: bool = false) : bool*
- *joj (val: JsonValue?; id: string) : JsonValue?*
- *jon (val: JsonValue?; id: string; defVal: double = 0lf) : double*
- *jos (val: JsonValue?; id: string; defVal: string = “”) : string*

`dap::j_s(val: JsonValue?; defVal: string = "") : string()`

Returns the string value of a JSON value, or *defVal* if not a string.

Arguments

- **val** : *JsonValue*?
- **defVal** : string

`dap::job(val: JsonValue?; id: string; defVal: bool = false) : bool()`

Returns a boolean JSON field by name, or *defVal* if not found.

Arguments

- **val** : *JsonValue*?
- **id** : string
- **defVal** : bool

`dap::joj(val: JsonValue?; id: string) : JsonValue?()`

Returns a nested JSON object field by name, or null if not found.

Arguments

- **val** : *JsonValue*?
- **id** : string

`dap::jon(val: JsonValue?; id: string; defVal: double = 0lf) : double()`

Returns a numeric JSON field by name, or *defVal* if not found.

Arguments

- **val** : *JsonValue*?
- **id** : string
- **defVal** : double

`dap::jos(val: JsonValue?; id: string; defVal: string = "") : string()`

Returns a string JSON field by name, or *defVal* if not found.

Arguments

- **val** : *JsonValue*?
- **id** : string
- **defVal** : string

13.9 Heartbeat callback injection

The HEARTBEAT module provides heartbeat callback injection. It automatically inserts a `heartbeat()` call at the beginning of every function body and at the start of every `for` / `while` loop body. This allows long-running scripts to periodically yield control for cooperative multitasking or watchdog checks.

All functions and symbols are in “heartbeat” module, use require to get access to it.

```
require daslib/heartbeat
```

Note: This module uses `[infer_macro]` to instrument code at compile time. The `heartbeat()` function must be defined in the calling context.

CHAPTER

FOURTEEN

CODE QUALITY AND REFACTORING

Linting, code validation, refactoring tools, and source formatting.

14.1 Paranoid lint pass

The LINT module implements static analysis checks for daslang code. It provides customizable lint rules that detect common mistakes, style violations, and potential bugs at compile time.

See also *Global lint pass (paranoid checks)* for applying lint diagnostics to all modules.

All functions and symbols are in "lint" module, use require to get access to it.

```
require daslib/lint
```

14.1.1 Lint operations

- *paranoid (prog: ProgramPtr; compile_time_errors: bool)*

`lint::`**`paranoid`**(*prog: ProgramPtr; compile_time_errors: bool*)

Runs the paranoid lint visitor on the program to check for common coding issues.

Arguments

- **prog** : *ProgramPtr*
- **compile_time_errors** : bool

14.2 Global lint pass (paranoid checks)

The LINT_EVERYTHING module applies paranoid lint diagnostics to all modules. Like `lint`, but applies paranoid diagnostics to every module in the program, not just the one that requires it.

All functions and symbols are in "lint_everything" module, use require to get access to it.

```
require daslib/lint_everything
```

Note: This module uses `[global_lint_macro]` to enable paranoid checks across all modules in the compilation unit. Any lint warnings will be treated as compilation errors.

14.3 Code validation annotations

The VALIDATE_CODE module implements AST validation passes that check for common code quality issues, unreachable code, missing return statements, and other semantic errors beyond what the type checker verifies.

All functions and symbols are in “validate_code” module, use require to get access to it.

```
require daslib/validate_code
```

14.3.1 Function annotations

validate_code::**verify_completion**

Verify that the function completes without infinite loops or recursion This annotation checks for potential infinite loops and recursive calls within the annotated function. If any are detected, a compile-time error is generated, preventing the code from compiling.

14.4 Automated refactoring tools

The REFACTOR module implements automated code refactoring transformations. It provides tools for renaming symbols, extracting functions, and other structural code changes that preserve program semantics.

All functions and symbols are in “refactor” module, use require to get access to it.

```
require daslib/refactor
```

14.4.1 Function annotations

refactor::**ExtractMethodMacro**

Function annotation implementing extract-method refactoring.

refactor::**ExtractVariableFunction**

Function annotation for extract-variable target functions.

14.4.2 Call macros

refactor::**extract_variable**

Call macro implementing extract-variable refactoring.

14.4.3 Refactoring operations

- *extract_expression (method_name: string; expr: auto) : auto*
- *extract_method (method_name: string; blk: block<():void>)*
- *extract_variable_nonref (method_name: string; expr: auto) : auto*
- *extract_variable_ref (method_name: string; var expr: auto(TT)& ==const) : TT&*

`refactor::`**`extract_expression(method_name: string; expr: auto) : auto`**`()`

Marks an expression for expression extraction refactoring.

Arguments

- **method_name** : string
- **expr** : auto

`refactor::`**`extract_method`**`(`*method_name: string; blk: block<():void>*`)`

Marks a block of code for method extraction refactoring.

Arguments

- **method_name** : string
- **blk** : block<void>

`refactor::`**`extract_variable_nonref(method_name: string; expr: auto) : auto`**`()`

Marks an expression for variable extraction by value.

Arguments

- **method_name** : string
- **expr** : auto

`refactor::`**`extract_variable_ref(method_name: string; expr: auto(TT)& ==const) : TT&`**`()`

Marks an expression for variable extraction by reference.

Arguments

- **method_name** : string
- **expr** : auto(TT)&!

14.5 Consume argument optimization

The CONSUME module implements the `consume` pattern, which moves ownership of containers and other moveable values while leaving the source in a default-constructed state. This enables efficient ownership transfer.

All functions and symbols are in “consume” module, use require to get access to it.

```
require daslib/consume
```

14.5.1 Function annotations

consume::**consume**

This annotation ensures that all arguments to the function are passed as moved values. For example [consume(a,b)] ensures that both a and b are passed as moved values.

14.6 Call argument removal annotation

The REMOVE_CALL_ARGS module provides AST transformation macros that remove specific arguments from function calls at compile time. Used for implementing optional parameter patterns and compile-time argument stripping.

All functions and symbols are in "remove_call_args" module, use require to get access to it.

```
require daslib/remove_call_args
```

14.6.1 Function annotations

remove_call_args::**remove_call_args**

This macro removes all arguments by given indices [remove_call_args(arg=(1,2,3))]

14.7 Source code formatter

The DAS_SOURCE_FORMATTER module implements source code formatting for daslang. It can parse and re-emit daslang source code with consistent indentation, spacing, and line breaking rules. Used by editor integrations and code quality tools.

All functions and symbols are in "das_source_formatter" module, use require to get access to it.

```
require daslib/das_source_formatter
```

14.7.1 Formatting

- *format_source (file_data: array<uint8>) : string*
- *format_source_string (file_data: string const&) : string*

das_source_formatter::**format_source(file_data: array<uint8>) : string**()

Formats daslang source code given as a byte array and returns the formatted result.

Arguments

- **file_data** : array<uint8> implicit

das_source_formatter::**format_source_string(file_data: string const&) : string**()

Formats a daslang source code string and returns the formatted result.

Arguments

- **file_data** : string& implicit

14.8 File-based source code formatter

The DAS_SOURCE_FORMATTER_FIO module extends the source formatter with file I/O capabilities, enabling formatting of daslang source files on disk. It reads, formats, and writes back source files in place or to new locations.

All functions and symbols are in "das_source_formatter_fio" module, use require to get access to it.

```
require daslib/das_source_formatter_fio
```

14.8.1 File formatting

- *format_file (file_name: string)*
- *format_files (file_names: array<string>)*

`das_source_formatter_fio::`**`format_file`**(*file_name: string*)

Reads a daslang source file, formats it, and writes the result back if changed.

Arguments

- **file_name** : string

`das_source_formatter_fio::`**`format_files`**(*file_names: array<string>*)

Formats multiple daslang source files in place.

Arguments

- **file_names** : array<string>