

Odin Language Overview

Ginger Bill and the Odin contributors
Oct 14, 2026

Introduction

This article is an introduction to the Odin Programming Language. You're assumed to have basic knowledge of programming concepts such as variables, statements, and types. You will see how familiar things are done in Odin, while also being introduced to new concepts.

If you haven't already installed Odin, then you can read how in the [Getting Started with Odin](#) guide.

Hellope!

To begin this tour, let us start with a modified version of the famous "hello world" program:

```
package main

import "core:fmt"

main :: proc() {
    fmt.println("Hellope!")
}
```

Save this code to a `.odin` file, then compile and run it using `odin run <dir>`. For the current directory:

```
odin run .
```

The `run` command compiles the `.odin` file to an executable and then runs that executable after compilation. If you do not wish to run the executable after compilation, the `build` command can be used.

```
odin build <dir>
```

Odin thinks in terms of directory-based packages. The `odin build <dir>` command takes all the files in the directory `<dir>`, compiles them into a package and then turns that into an executable. You can also tell it to treat a single file as a complete package, by adding `-file`, like so:

```
odin run hellope.odin -file
```

Variable declarations

A variable declaration declares a new variable for the current scope.

```
x: int // declares `x` to have type `int`
y, z: int // declares `y` and `z` to have type `int`
```

Variables are initialized to zero by default unless specified otherwise.

Note

Declarations have to be *unique* within a scope.

```
x := 10
x := 20 // Redeclaration of `x` in this scope
y, z := 20, 30
test, z := 20, 30 // not allowed since `z` exists already
```

Assignment statements

The assignment statement assigns a new value to a variable/location:

```
x: int = 123 // declares a new variable `x` with type `int` and assigns a value to it
x = 637 // assigns a new value to `x`
```

= is the assignment operator.

You can assign multiple variables with it:

```
x, y := 1, "hello" // declares `x` and `y` and infers the types from the assignments
y, x = "bye", 5
```

Note

:= is two tokens, : and =. The following are all equivalent:

```
x: int = 123
x:      = 123 // default type for an integer literal is `int`
x := 123
```

Literals

String and character literals

String literals are enclosed in double quotes and character literals in single quotes. Special characters are escaped with a backslash \.

```
"This is a string"
'A'
'\n' // newline character
"C:\\Windows\\notepad.exe"
```

Raw string literals are enclosed in single back ticks.

```
`C:\\Windows\\notepad.exe`
```

The length of a string can be found using the built-in `len` proc:

```
len("Foo")
len(some_string)
```

If the string passed to `len` is a compile-time constant, the value from `len` will be a compile-time constant.

Multi-line string literals

Multi-line string literals are enclosed in triple quotes: `"""` processes escape sequences like a normal string literal, and ````` is raw like a back tick string literal.

The text must begin on the line after the opening delimiter and the closing delimiter must be on its own line.

The indentation of the closing delimiter is removed from every line, which allows the literal to be indented along with the surrounding code. The newline before the closing delimiter is not part of the string.

```
greeting := """
  Helloe,
  "World"!
  """
assert(greeting == "Helloe,\n \"World\"!")

path := ```
  C:\\Windows\\notepad.exe
  ```
assert(path == `C:\\Windows\\notepad.exe`)
```

Every non-blank line must start with the same indentation as the closing delimiter, otherwise it is a compile error.

### Escape characters

- `\a` - bell (BEL)
- `\b` - backspace (BS)
- `\e` - escape (ESC)
- `\f` - form feed (FF)
- `\n` - newline
- `\r` - carriage return

- `\t` - tab
- `\v` - vertical tab (VT)
- `\\` - backslash
- `\"` - double quote (if needed)
- `\'` - single quote (if needed)
- `\NNN` - octal 6 bit character (3 digits)
- `\xNN` - hexadecimal 8 bit character (2 digits)
- `\uNNNN` - hexadecimal 16-bit Unicode character UTF-8 encoded (4 digits)
- `\UNNNNNNNN` - hexadecimal 32-bit Unicode character UTF-8 encoded (8 digits)

## Numbers

Numerical literals are written similar to most other programming languages. A useful feature in Odin is that underscores are allowed for better readability: `1_000_000_000` (one billion). A number that contains a dot is a floating point literal: `1.0e9` (one billion). If a number literal is suffixed with `i`, it is an imaginary number literal: `2i` (2 multiply the square root of -1). The suffixes `j` and `k` are the other two imaginary units of a [quaternion](#): `1 + 2i + 3j + 4k`.

Binary literals are prefixed with `0b`, octal literals with `0o`, and hexadecimal literals with `0x`. A leading zero does not produce an octal constant (unlike C).

Hexadecimal float literals are prefixed with `0h` and specify the exact bit pattern of an IEEE-754 floating point value. The number of hexadecimal digits determines how the bits are interpreted: 4 digits for `f16`, 8 digits for `f32`, and 16 digits for `f64`.

```
one_f16: f16 = 0h3c00
one_f32: f32 = 0h3f80_0000
one_f64: f64 = 0h3ff0_0000_0000_0000
inf: f32 = 0h7f80_0000
```

In Odin, if a number constant can be represented by a type without precision loss, it will automatically convert to that type.

```
x: int = 1.0 // A float literal but it can be represented by an integer without precision loss
```

Constant literals are "untyped" which means that they can implicitly convert to a type.

```
x: int // `x` is typed as being of type `int`
x = 1 // `1` is an untyped integer literal which can implicitly convert to `int`
```

## Comments

Comments can be anywhere outside of a string or character literal. Single line comments begin with `//`:

```
// A comment

my_integer_variable: int // A comment for documentation
```

Multi-line comments begin with `/*` and end with `*/`. Multi-line comments can also be nested (unlike in C):

```
/*
 You can have any text or code here and
 have it be commented.
 */
 NOTE: comments can be nested!
 */
*/
```

Comments are parsed as tokens within the compiler. This allows `odin doc` to use the comments directly preceding a declaration as its documentation, such as the documentation on [pkg.odin-lang.org](http://pkg.odin-lang.org).

## Constant declarations

Constants are entities (symbols) which have an assigned value. The constant's value cannot be changed. The constant's value must be able to be evaluated at compile time:

```
x :: "what" // constant `x` has the untyped string value "what"
```

Constants can be explicitly typed like a variable declaration:

```
y : int : 123
z :: y + 7 // constant computations are possible
```

Constants are not limited to numbers, booleans, and strings (like some languages). Compound literals of structs, arrays, and enumerated arrays can be constants too, and they can be accessed, indexed, and sliced at compile time:

```
Vector2 :: struct {x, y: f32}
Entity :: struct {id: int, using position: Vector2}

PLAYER :: Entity{id = 1, position = {3, 4}}
#assert(PLAYER.id == 1)
#assert(PLAYER.y == 4) // `using` works as expected

Level :: enum {Easy, Normal, Hard, Expert}
LIVES :: [Level]int{.Easy .= .Hard = 3, .Expert = 1}
#assert(LIVES[.Normal] == 3)

GREETING :: "Hellope!"
#assert(GREETING[0] == 'H')
#assert(GREETING[0:5] == "Hello")
```

For more information regarding value declarations in general, please see the Odin FAQ and Ginger Bill's article [On the Aesthetics of the Syntax of Declarations](#).

## Packages

Odin programs consist of packages. A package is a directory of Odin code files, all of which have the same package declaration at the top. The `odin build <dir>` command takes all the files in the directory `<dir>`, compiles them into a package and then turns that into an executable. Execution starts in the package's `main` procedure.

You can also tell the compiler to treat a single file as a complete package, by adding `-file`, like so:

```
odin run hellope.odin -file
```

### import statement

The following program imports the `fmt` and `os` packages from the `core` library collection.

```
package main

import "core:fmt"
import "core:os"

main :: proc() {
}
```

The `core:` prefix is used to state where the import is meant to look; this is called a library collection. If no prefix is present, the import will look relative to the current file.

#### Note

By convention, the package name is the same as the last element in the import path. The `"core:fmt"` package comprises files that begin with the statement `package fmt`. However, this is not enforced by the compiler, which means the default name for the import name will be determined by the last element in the import path if possible.

A different import name can be used over the default package name:

```
import "core:fmt"
import foo "core:fmt" // reference a package by a different name
```

### Exported names

All declarations in a package are public by default.

The `private` attribute can be applied to an entity to prevent it from being exported from a package.

```
@(private)
my_variable: int // cannot be accessed outside this package
```

You may also make an entity private to *the file* instead of the package.

```
@(private="file")
my_variable: int // cannot be accessed outside this file
```

`@(private)` is equivalent to `@(private="package")`.

## Authoring a package

A package is a directory of Odin code files, all of which have the same package declaration at the top, e.g. `package main`. Each .odin file *must* have the same package name. A directory cannot contain more than 1 package.

## Organizing packages

Packages may be thematically organized by placing them in subdirectories of another package. For example: `core:image/png` and `core:image/tga`, as subdirectories of `core:image`. Nesting these packages is a helpful taxonomy. It does not imply a dependency: `core:foo/bar` does not need to import `core:foo` and reference anything from it.

# Control flow statements

## for statement

Odin has only one loop statement, the `for` loop.

### Basic for loop

A basic `for` loop has three components separated by semicolons:

- The initial statement: executed before the first iteration
- The condition expression: evaluated before every iteration
- The post statement: executed at the end of every iteration

The loop will stop executing when the condition is evaluated to `false`.

```
for i := 0; i < 10; i += 1 {
 fmt.Println(i)
}
```

### Note

Unlike other languages like C, there are no parentheses `( )` surrounding the three components. Braces `{ }` or a `do` are always required.

```
for i := 0; i < 10; i += 1 { }
for i := 0; i < 10; i += 1 do single_statement()
```

The initial and post statements are optional:

```
i := 0
for ; i < 10; {
 i += 1
}
```

These semicolons can be dropped. This `for` loop is equivalent to C's `while` loop:

```
i := 0
for i < 10 {
 i += 1
}
```

If the condition is omitted, this produces an infinite loop:

```
for {
}
```

## Range-based for loop

The basic `for` loop

```
for i := 0; i < 10; i += 1 {
```

```
 fmt.println(i)
}
```

can also be written

```
for i in 0..10 {
 fmt.println(i)
}
// or
for i in 0..9 {
 fmt.println(i)
}
```

where  $a..b$  denotes a closed interval  $[a, b]$ , i.e. the upper limit is *inclusive*, and  $a..<b$  denotes a half-open interval  $[a, b)$ , i.e. the upper limit is *exclusive*.

Certain built-in types can be iterated over:

```
some_string := "Hello, []"
for character in some_string {
 fmt.println(character)
}

some_array := [3]int{1, 4, 9}
for value in some_array {
 fmt.println(value)
}

some_slice := []int{1, 4, 9}
for value in some_slice {
 fmt.println(value)
}

some_dynamic_array := [dynamic]int{1, 4, 9} // must be enabled with `#+feature dynamic-literals`
defer delete(some_dynamic_array)
for value in some_dynamic_array {
 fmt.println(value)
}

some_map := map[string]int{"A" = 1, "C" = 9, "B" = 4} // must be enabled with `#+feature dynamic-literals`
defer delete(some_map)
for key in some_map {
 fmt.println(key)
}
```

Alternatively a second index value can be added:

```
for character, index in some_string {
 fmt.println(index, character)
}
for value, index in some_array {
 fmt.println(index, value)
}
for value, index in some_slice {
 fmt.println(index, value)
}
for value, index in some_dynamic_array {
 fmt.println(index, value)
}
for key, value in some_map {
 fmt.println(key, value)
}
```

The iterated values are *copies* and cannot be written to.

When iterating a string, the characters will be *rune*s and not bytes. `for in` assumes the string is encoded as UTF-8.

```
str: string = "Some text"
for character in str {
 assert(type_of(character) == rune)
 fmt.println(character)
}
```

You can iterate arrays and slices by-reference with the [address operator](#):

```
for &value in some_array {
 value = something
}
for &value in some_slice {
 value = something
}
for &value in some_dynamic_array {
 value = something
}
// does not impact the second index value
for &value, index in some_dynamic_array {
```

```
 value = something
}
```

Map values can be iterated by-reference, but their keys cannot since map keys are immutable:

```
some_map := map[string]int{"A" = 1, "C" = 9, "B" = 4} // must be enabled with `#+feature dynamic-literals`
defer delete(some_map)

for key, &value in some_map {
 value += 1
}

fmt.Println(some_map["A"]) // 2
fmt.Println(some_map["C"]) // 10
fmt.Println(some_map["B"]) // 5
```

### Note

It is not possible to iterate a string in a by-reference manner as strings are immutable.

Range-based `for` loops can also iterate over the results of a procedure that returns a multi-valued tuple whose last value is a `bool`. The loop continues while the boolean is `true`, and the preceding values in the tuple are available as loop variables. This is similar in concept to a "custom iterator" but is just syntactic sugar for calling the procedure in a loop.

```
iter_count := 0
some_procedure :: proc() -> (int, int, int, cond: bool) {
 iter_count += 1
 if iter_count < 4 {
 return iter_count, iter_count*2, iter_count*3, true
 } else {
 return 0, 0, 0, false
 }
}

for a, b, c in some_procedure() {
 fmt.Println(a, b, c)
 // Prints
 // 1 2 3
 // 2 4 6
 // 3 6 9
}
```

### for reverse iteration

The `#reverse` directive makes a range-based `for` loop iterate in reverse.

```
array := [?]int { 10, 20, 30, 40, 50 }

#reverse for x in array {
 fmt.Println(x) // 50 40 30 20 10
}
```

### for loop unrolling

The `#unroll` directive takes a `for` loop and expands it at compile-time to the individual statements, repeated for as many times as the loop would normally iterate. This may result in performance improvements or provides the ability to repeat a set of instructions a limited number of times without explicitly writing each out in repetition.

Please note that `#unroll` may only be used with ranged `for` loops that have constant intervals known at compile-time.

```
x: [4]u8 = 0xFF
y: [4]u8 = 0x88
#unroll for i in 0..<len(x) {
 x[i] ^= y[i]
}
```

### if statement

Odin's `if` statements do not need to be surrounded by parentheses `( )` but braces `{ }` or `do` are required.

```
if x >= 0 {
 fmt.Println("x is positive")
}
```

Like `for`, the `if` statement can start with an initial statement to execute before the condition. Variables declared by the initial statement are only in the scope of that `if` statement.

```
if x := foo(); x < 0 {
 fmt.Println("x is negative")
}
```

Variables declared inside an `if` initial statement are also available to any of the `else` blocks:

```
if x := foo(); x < 0 {
 fmt.Println("x is negative")
} else if x == 0 {
 fmt.Println("x is zero")
} else {
 fmt.Println("x is positive")
}
```

## switch statement

A switch statement is another way to write a sequence of if-else statements. In Odin, the default case is denoted as a case without any expression.

```
switch arch := ODIN_ARCH; arch {
case .i386, .wasm32, .arm32:
 fmt.Println("32 bit")
case .amd64, .wasm64p32, .arm64, .riscv64:
 fmt.Println("64 bit")
case .Unknown:
 fmt.Println("Unknown architecture")
}
```

Odin's `switch` is like the one in C or C++, except that Odin only runs the selected case. This means that a `break` statement is not needed at the end of each case. Another important difference is that the case values need not be integers nor constants.

To achieve a C-like fall through into the next case block, the keyword `fallthrough` can be used.

Switch cases are evaluated from top to bottom, stopping when a case succeeds. For example:

```
switch i {
case 0:
case foo():
}
```

`foo()` does not get called if `i==0`. If all the case values are constants, the compiler may optimize the switch statement into a jump table (like C).

A switch statement without a condition is the same as `switch true`. This can be used to write a clean and long if-else chain and have the ability to `break` if needed

```
switch {
case x < 0:
 fmt.Println("x is negative")
case x == 0:
 fmt.Println("x is zero")
case:
 fmt.Println("x is positive")
}
```

A switch statement can also use ranges like a range-based loop:

```
switch c {
case 'A'..'Z', 'a'..'z', '0'..'9':
 fmt.Println("c is alphanumeric")
}

switch x {
case 0..<10:
 fmt.Println("units")
case 10..<13:
 fmt.Println("pre-teens")
case 13..<20:
 fmt.Println("teens")
case 20..<30:
 fmt.Println("twenties")
}
```

## #partial switch

With `enum` values:

```

Foo :: enum {
 A,
 B,
 C,
 D,
}

f := Foo.A
switch f {
case .A: fmt.Println("A")
case .B: fmt.Println("B")
case .C: fmt.Println("C")
case .D: fmt.Println("D")
case: fmt.Println("?")
}

#partial switch f {
case .A: fmt.Println("A")
case .D: fmt.Println("D")
}

```

With union types (see [Type switch statement](#))

```

Foo :: union {int, bool}
f: Foo = 123
switch _ in f {
case int: fmt.Println("int")
case bool: fmt.Println("bool")
case:
}

#partial switch _ in f {
case bool: fmt.Println("bool")
}

```

## defer statement

A defer statement defers the execution of a statement until the end of the scope it is in.

The following will print 4 then 234:

```

package main

import "core:fmt"

main :: proc() {
 x := 123
 defer fmt.Println(x)
 {
 defer x = 4
 x = 2
 }
 fmt.Println(x)

 x = 234
}

```

You can defer an entire block too:

```

{
 defer {
 foo()
 bar()
 }
 // This is equivalent to `defer { if cond { bar() } }` because the `if` is
 // a statement in its own right.
 defer if cond {
 bar()
 }
}

```

Defer statements are executed in the reverse order that they were declared:

```

defer fmt.Println("1")
defer fmt.Println("2")
defer fmt.Println("3")

```

Will print 3, 2, and then 1.

A real world use case for defer may be something like the following:

```

f, err := os.Open("my_file.txt")
if err != nil {
 // handle error
}
defer os.Close(f)
// rest of code

```

In this case, it acts akin to an explicit C++ destructor however, the error handling is basic control flow. It's important to note that `defer` cannot be used to change a procedure's named return values, as it runs after exit when the values have already been returned.

```
foo :: proc() -> (n: int) {
 defer {
 n = 456 // This won't affect `n`
 }
 n = 123
 return
}
```

### Note

The `defer` construct in Odin differs from Go's `defer`, which is function-exit and relies on a closure stack system.

### when statement

The `when` statement is almost identical to the `if` statement but with some differences:

- Each condition must be a constant expression as a `when` statement is evaluated at compile time.
- The statements within a branch do not create a new scope
- The compiler checks the semantics and code **only** for statements that belong to the first condition that is `true`
- An initial statement is not allowed in a `when` statement
- `when` statements are allowed at file scope

Example:

```
when ODIN_ARCH == .i386 {
 fmt.println("32 bit")
} else when ODIN_ARCH == .amd64 {
 fmt.println("64 bit")
} else {
 fmt.println("Unsupported architecture")
}
```

The `when` statement is very useful for writing platform specific code. This is akin to the `#if` construct in C's pre-processor. However, in Odin, it is type checked.

See the [Conditional compilation section](#) for examples of built-in constants you can use with `when` statements.

## Branch statements

### break statement

A for loop, conditional, or a switch statement can be left prematurely with a `break` statement. It leaves the innermost construct, unless a label of a construct is given:

```
for cond {
 switch {
 case:
 if cond {
 break // break out of the `switch` statement
 }
 }

 break // break out of the `for` statement
}

loop: for cond1 {
 for cond2 {
 break loop // leaves both loops
 }
}

outer: if cond {
 ok := check_something()
 if !ok {
 break outer // label names are required with conditionals
 }
}

exit: {
 if true {
 break exit // works with labeled blocks too
 }
 fmt.println("This line will never print.")
}
```

```
}
```

### continue statement

As in many programming languages, a `continue` statement starts the next iteration of a loop prematurely:

```
for cond {
 if get_foo() {
 continue
 }
 fmt.println("Hellope")
}
```

### fallthrough statement

Odin's `switch` is like the one in C or C++, except that Odin only runs the selected case. This means that a `break` statement is not needed at the end of each case. Another important difference is that the case values need not be integers nor constants.

`fallthrough` can be used to explicitly fall through into the next case block:

```
switch i {
case 0:
 foo()
 fallthrough
case 1:
 bar()
}
```

## Procedures

In Odin, a procedure is something that can do work, which some languages call *functions* or *methods*. A procedure literal in Odin is defined with the `proc` keyword:

```
fibonacci :: proc(n: int) -> int {
 switch {
 case n < 1:
 return 0
 case n == 1:
 return 1
 }
 return fibonacci(n-1) + fibonacci(n-2)
}

fmt.println(fibonacci(3)) // 2
```

For more information regarding value declarations in general, please see the Odin FAQ.

### Parameters

Procedures can take zero or many parameters. The following example is a basic procedure that multiplies two integers together:

```
multiply :: proc(x: int, y: int) -> int {
 return x * y
}

fmt.println(multiply(137, 432))
```

When two or more consecutive parameters share a type, you can omit the other types from previous names, like with variable declarations. In this example: `x: int, y: int` can be shortened to `x, y: int`, for example:

```
multiply :: proc(x, y: int) -> int {
 return x * y
}

fmt.println(multiply(137, 432))
```

By default, Odin procedures use the "odin" calling convention. This calling convention is the same as C, however it differs in a couple of ways:

- It promotes values to a pointer if that's more efficient on the target system, and
- It includes a pointer to the current context as an implicit additional argument.

The promotion is enabled by the fact that all parameters are immutable in Odin, and its rules are consistent for a given type and platform and can be relied on since they are part of the calling convention.

Passing a pointer value makes a copy of the pointer, not the data it points to. Slices, dynamic arrays, and maps have no special considerations here; they are normal structures with pointer fields, and are passed as such. Their elements will not be copied.

### Shadowing parameters

To mutate the procedure parameter (like in C), an explicit copy is required. This can be done through shadowing the variable declaration:

```
foo :: proc(x: int) {
 x := x // explicit mutation
 for x > 0 {
 fmt.println(x)
 x -= 1
 }
}
```

### Variadic parameters

Procedures can be variadic, taking a varying number of arguments:

```
sum :: proc(nums: ..int) -> (result: int) {
 for n in nums {
 result += n
 }
 return
}
fmt.println(sum()) // 0
fmt.println(sum(1, 2)) // 3
fmt.println(sum(1, 2, 3, 4, 5)) // 15

odds := []int{1, 3, 5}
fmt.println(sum(..odds)) // 9, passing a slice as varargs
```

### Multiple results

A procedure in Odin can return any number of results. For example:

```
swap :: proc(x, y: int) -> (int, int) {
 return y, x
}
a, b := swap(1, 2)
fmt.println(a, b) // 2 1
```

A call that returns multiple values can be passed directly as multiple arguments to another procedure, including into a variadic parameter, or used as multiple elements of a [compound literal](#):

```
sum :: proc(nums: ..int) -> (result: int) {
 for n in nums {
 result += n
 }
 return
}
fmt.println(sum(swap(1, 2))) // 3
fmt.println(sum(swap(1, 2), 3)) // 6

Pair :: struct {x, y: int}
p := Pair{swap(1, 2)} // Pair{x = 2, y = 1}
xs := []int{swap(1, 2), 3} // [2, 1, 3]
```

### Named results

Return values in Odin may be named. If so, they are treated as variables defined at the top of the procedure, like input parameters. A `return` statement without arguments returns the named return value (or values).

"Naked" return statements should only be used in short procedures as it reduces clarity when reading.

```
do_math :: proc(input: int) -> (x, y: int) {
 x = 2*input + 1
 y = 3*input / 5
 return x, y
}
do_math_with_naked_return :: proc(input: int) -> (x, y: int) {
 x = 2*input + 1
 y = 3*input / 5
 return // A "naked" return statement, as no values are explicitly specified.
}
```

Assigning [default values](#) to named results is also possible:

```
conditionally_blue :: proc(red: bool) -> (color := "blue") {
 if red {
 return "red"
 }
 return
}
```

## Named arguments

When calling a procedure, it is not clear in which order parameters might appear. Therefore, the arguments can be named, like a struct literal, to make it clear which argument a parameter is for:

```
create_window :: proc(title: string, x: int, y: int, width: int, height: int, monitor: ^Monitor) -> (^Window, Window_Error) {...}
window, err := create_window(title="Hellope Title", monitor=nil, width=854, height=480, x=0, y=0)
```

Named and positional arguments can be mixed. This is often useful when a procedure has a lot of arguments *or* you want to customize [default values](#).

**Positional** arguments are not allowed after **named** arguments.

```
foo :: proc(value: int, name: string, x: bool, y: f32, z := 0) { }
foo(134, "hellope", x=true, y=4.5)
```

## Default values

The `create_window` procedure may be easier to use if default values are provided, which will be used if they are not specified:

```
create_window :: proc(title: string, x := 0, y := 0, width := 854, height := 480, monitor: ^Monitor = nil) -> (^Window, Window_Error) {...}
window1, err1 := create_window("Title1")
window2, err2 := create_window(title="Title1", width=640, height=360)
```

Default values are assigned at the start of the procedure call and can be [overwritten](#). They may also be assigned to [named results](#).

### Note

These default values must be compile time known values, such as a constant value or `nil` (if the type supports it).

## Explicit procedure overloading

Unlike other languages, Odin provides the ability to explicitly overload procedures:

```
bool_to_string :: proc(b: bool) -> string {...}
int_to_string :: proc(i: int) -> string {...}

to_string :: proc{bool_to_string, int_to_string}
```

### Rationale behind explicit overloading

The design goals of Odin were explicitness and simplicity. Implicit procedure overloading complicates the scoping system. In C++, you cannot nest procedures within procedures, so all procedure look-ups are done at the global scope. In Odin, procedures can be nested within procedures and, as a result, determining which procedure should be used, in the case of implicit overloading, is complex.

Explicit overloading has many advantages:

- Explicitness of what is overloaded
- Able to refer to the specific procedure if needed
- Clear which scope the entity name belongs to
- Ability to specialize parametric polymorphic procedures if necessary, which have the same parameter but different bounds (see [where clauses](#))

```
foo :: proc{
 foo_bar,
 foo_baz,
 foo_baz2,
 another_thing_entirely,
```

}

## Basic types

Odin's basic types are:

```
bool b8 b16 b32 b64 // booleans

// integers
int i8 i16 i32 i64 i128
uint u8 u16 u32 u64 u128 uintptr

// endian specific integers
i16le i32le i64le i128le u16le u32le u64le u128le // little endian
i16be i32be i64be i128be u16be u32be u64be u128be // big endian

f16 f32 f64 // floating point numbers

// endian specific floating point numbers
f16le f32le f64le // little endian
f16be f32be f64be // big endian

complex32 complex64 complex128 // complex numbers

quaternion64 quaternion128 quaternion256 // quaternion numbers

rune // signed 32 bit integer
 // represents a Unicode code point
 // is a distinct type to `i32`

// strings
string cstring // UTF-8 based strings
string16 cstring16 // UTF-16 based strings

// raw pointer type
rawptr

// runtime type information specific type
typeid
any
```

byte is an alias for u8.

The `uintptr` type is pointer sized, and the `int`, `uint` types are the "natural" register size, which is guaranteed to be greater than or equal to the size of a pointer (i.e. `size_of(uint) >= size_of(uintptr)`). When you need an integer value, you should default to using `int` unless you have a specific reason to use a sized or unsigned integer type.

### Note

The Odin `string` type stores the pointer to the data and the length of the string. `cstring` is used to interface with foreign libraries written in/for C that use zero-terminated strings. `string16` and `cstring16` are their UTF-16 equivalents.

## Zero values

Variables declared without an explicit initial value are given their *zero* value.

The zero value is:

- `0` for numeric and rune types
- `false` for boolean types
- `""` (the empty string) for strings
- `nil` for pointer, `typeid`, and any types.

The expression `{}` can be used as the zero value of any type. This is not recommended when a type has a specific zero value shown above, as that is clearer; please prefer that.

## Type conversion

The expression `T(v)` converts the value `v` to the type `T`.

```
i: int = 123
f: f64 = f64(i)
u: u32 = u32(f)
```

or with type inference:

```
i := 123
f := f64(i)
u := u32(f)
```

Unlike C, assignments between values of a different type require an explicit conversion.

### Cast operator

The `cast` operator can also be used to do the same thing:

```
i := 123
f := cast(f64)i
u := cast(u32)f
```

This is useful in some contexts but has the same semantic meaning.

Conversions also apply element-wise to fixed length arrays, see [array casts](#).

### Transmute operator

The `transmute` operator is a bit cast conversion between two types of the same size:

```
f := f32(123)
u := transmute(u32)f
```

This is akin to doing the following pointer cast manipulations:

```
f := f32(123)
u := (^u32)(&f)^
```

However, `transmute` does not require taking the address of the value in question, which may not be possible for many expressions.

### Untyped types

In the Odin type system, certain expressions will have an "untyped" type. An untyped type can implicitly convert to a "typed" type.

```
I :: 42 // untyped integer, will implicitly convert to any numeric type (int, u32, f64, quaternion128 etc)
F :: 1.37 // untyped float, will implicitly convert to any numeric type that can support fractional parts (f64, quaternion128 etc)
S :: "Hellope" // untyped string, will implicitly convert to string and cstring
B :: true // untyped boolean, will implicitly convert to bool, b8, b16, etc.
```

(The more formal name for these "untyped" types is [existential](#) or [abstract](#) types.)

### Auto cast operation

The `auto_cast` operator automatically casts an expression to the destination's type if possible:

```
x: f32 = 123
y: int = auto_cast x
```

**Note:** This operation is only recommended to be used for prototyping and quick tests. Please do not abuse it.

### Built-in constants, values, and procedures

There are a few built-in constants and values in Odin which have different uses:

```
false // untyped boolean constant equivalent to the expression 0!=0
true // untyped boolean constant equivalent to the expression 0==0
nil // untyped nil value used for certain values
--- // untyped undefined value used to explicitly not initialize a variable
```

`---` is useful if you want to explicitly not initialize a variable with any default value:

```
x: int // initialized with its zero value
y: int = --- // uses uninitialized memory
```

This is the default behaviour in C, whilst the default behaviour in Odin is to zero the memory.

**Note**

--- is not a contract in that all memory is uninitialized, but that it may be. For example, as a side-effect of an implementation detail, padding in a struct must be zeroed to permit trivial bitwise memory comparison.

**Built-in procedures**

For the full list of builtin-procedures, see the documentation for [package builtin](#).

There are two kinds of built-in procedures in Odin:

- Compiler defined
- Core library defined

The compiler defined built-in procedures include:

| Procedure                                    | Description                                                                                    |
|----------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>len(x)</code>                          | The length of a string, array, slice, dynamic array, map, etc.                                 |
| <code>cap(x)</code>                          | The capacity of an array, slice, dynamic array, map, etc.                                      |
| <code>size_of(T)</code>                      | The size of a type in bytes                                                                    |
| <code>align_of(T)</code>                     | The alignment of a type in bytes                                                               |
| <code>offset_of(T, field)</code>             | The byte offset of a field within a struct                                                     |
| <code>offset_of(x.field)</code>              | The byte offset of a field within a struct, from a value                                       |
| <code>offset_of_by_string(T, "field")</code> | The byte offset of a field within a struct, by name                                            |
| <code>type_of(x)</code>                      | The type of an expression                                                                      |
| <code>typeid_of(T)</code>                    | The <a href="#">typeid</a> of a type                                                           |
| <code>type_info_of(T)</code>                 | The runtime type information of a type                                                         |
| <code>min(a, b, ...)</code>                  | The minimum of the values                                                                      |
| <code>max(a, b, ...)</code>                  | The maximum of the values                                                                      |
| <code>min(T), max(T)</code>                  | The smallest or largest value of a type, e.g. <code>min(i8)</code> , <code>max(My_Enum)</code> |
| <code>clamp(x, lo, hi)</code>                | Clamps a value between two bounds                                                              |
| <code>abs(x)</code>                          | The absolute value                                                                             |
| <code>swizzle(x, indices...)</code>          | See <a href="#">swizzle operations</a>                                                         |
| <code>complex, quaternion</code>             | See <a href="#">complex numbers and quaternions</a>                                            |
| <code>real, imag, jmag, kmag</code>          | See <a href="#">complex numbers and quaternions</a>                                            |
| <code>conj</code>                            | See <a href="#">complex numbers and quaternions</a>                                            |
| <code>expand_values(x)</code>                | See <a href="#">expand_values</a>                                                              |
| <code>compress_values(...)</code>            | See <a href="#">compress_values</a>                                                            |
| <code>soa_zip, soa_unzip</code>              | See <a href="#">soa_zip and soa_unzip</a>                                                      |
| <code>raw_data(x)</code>                     | See <a href="#">raw_data</a>                                                                   |
| <code>unreachable()</code>                   | Marks a point in the code which can never be reached                                           |

The core library defined built-in procedures live in [base:runtime](#) and include the memory and container procedures such as `new`, `make`, `delete`, `append`, `pop`, `clear`, `resize`, etc. (see [allocators](#) and [dynamic arrays](#)).

**[expand\\_values](#) and [compress\\_values](#)**

`expand_values` takes a struct or array value and expands it into multiple values, in field or element order. The prefix `**` operator is shorthand for `expand_values`:

```
Vector3 :: struct {x, y, z: f32}
v := Vector3{1, 4, 9}

x, y, z := expand_values(v)
x, y, z = **v // equivalent
```

```
// using `sum` from the multiple results section
arr := [3]int{7, 8, 9}
fmt.Println(sum(**arr)) // equivalent to `sum(arr[0], arr[1], arr[2])`
```

`compress_values` does the opposite: it packs multiple values (including the results of multi-valued calls) into a single struct or array value. If there is a type hint, that type is used. Otherwise, if all the values share the same type, the result is a fixed length array, else it is an anonymous struct:

```
v: Vector3 = compress_values(1, 4, 9) // Vector3{1, 4, 9}
a := compress_values(1, 2, 3) // [3]int{1, 2, 3}
s := compress_values(1, "hi", true) // struct{v0: int, v1: string, v2: bool}{1, "hi", true}
```

## Intrinsics

Lower level compiler-specific procedures and types are available in the `base:intrinsics` package. They are documented in the [package intrinsics](#) documentation, and include:

- Type introspection for use in [where clauses](#) and `when` statements
  - e.g. `type_is_integer(T)`, `type_elem_type(T)`, `type_has_field(T, "name")`, `type_field_bit_offset(T, "name")`, `type_canonical_name(T)`
- Bit operations
  - e.g. `count_ones`, `count_leading_zeros`, `reverse_bits`, `byte_swap`
- Checked and saturating arithmetic
  - e.g. `overflow_add`, `saturating_add`
- Memory operations
  - e.g. `mem_copy`, `mem_zero`, `volatile_load`, `unaligned_store`, `non_temporal_store`, `prefetch_read_data`
- Atomic operations
  - e.g. `atomic_load_explicit`, `atomic_compare_exchange_strong`, `atomic_thread_fence` ([core:sync](#) provides friendlier wrappers)
- SIMD operations
  - e.g. `simd_add`, `simd_shuffle`, `simd_reduce_add_ordered` ([core:simd](#) provides friendlier wrappers)
- Branch hints and miscellaneous procedures
  - e.g. `expect`, `likely`, `unlikely`, `trap`, `debug_trap`, `cpu_relax`, `read_cycle_counter`, `alloca`
- C variadic procedures support
  - `c_va_start`, `c_va_arg`, `c_va_end`, `c_va_copy` (see [Implementing C variadic procedures](#))

```
import "base:intrinsics"

Foo :: bit_field u16 {
 x: i32 | 3,
 y: u16 | 5,
}
#assert(intrinsics.type_field_bit_offset(Foo, "y") == 3)
#assert(intrinsics.type_field_bit_size(Foo, "y") == 5)

n := intrinsics.count_ones(u32(0b1011)) // 3
```

## string type

As previously mentioned the `odin string` type is just a `rawptr + len`.

The `core:strings` library was created to help dealing with string cloning, conversion of `string<->cstring` and other calls you find in standard libraries.

All procedures are [documented](#) and can be easily understood with code examples.

## string iteration

Iterating a string can be done in two ways - by runes or by bytes.

```
// by runes
x := "ABC"
for codepoint, index in x {
 fmt.Println(index, codepoint)
}
// 0 A
// 1 B
```

```

 // 2 C
}

// by bytes - string length is in bytes
for index in 0..<len(x) {
 fmt.Println(index, x[index])
 // 0 65
 // 1 66
 // 2 67
}

```

Iteration through runes is preferred since Odin strings are UTF-8 encoded. Core library procedures which work with individual bytes rather than runes are usually suffixed with `_byte`, e.g. `strings.index_byte` VS `strings.index_rune`.

### string format printing

The `core:fmt` library supports printing strings from byte arrays in structs, when additional tag information is supplied.

```

Foo :: struct {
 a: [L]u8 `fmt:"s"`, // whole buffer is a string
 b: [N]u8 `fmt:"s,0"`, // 0 terminated string
 c: [M]u8 `fmt:"q,n"`, // string with length determined by n, and use %q rather than %s
 n: int `fmt:"-n"`, // ignore this from formatting
}

```

### cstring type

The `cstring` type is a c-style string value, which is zero-terminated. It is equivalent to `char const *` in C. Its primary purpose is for easy interfacing with C. Please see the [foreign system](#) for more information.

A `cstring` is easily convertible to an Odin `string`. However, to convert a `string` to a `cstring`, it requires allocations if the value is not constant.

```

str: string = "Hellope"
cstr: cstring = "Hellope" // constant literal
cstr2 := string(cstr) // 0(n) conversion as it requires search from the zero-terminator
nstr := len(str) // 0(1)
ncstr := len(cstr) // 0(n)

```

### string16 and cstring16 types

`string16` and `cstring16` are the UTF-16 counterparts of `string` and `cstring`. They are primarily useful for interfacing with APIs which use UTF-16 strings, such as the Windows API (where `cstring16` is equivalent to `wchar_t const *`).

- `string16` stores a pointer to the `u16` data and its length. `len` returns the number of `u16` code units, not the number of characters.
- `cstring16` is a zero-terminated UTF-16 string.

A constant string literal implicitly converts to either type, being encoded as UTF-16 at compile time:

```

s: string16 = "Hellope"
c: cstring16 = "Hellope"
assert(len(s) == 7)

for r, i in s { // iterating a `string16` decodes the UTF-16 into runes
 fmt.Println(i, r)
}

emoji: string16 = "😄"
assert(len(emoji) == 2) // encoded as a surrogate pair

```

Like their UTF-8 equivalents, they can be converted to and from related types without allocation:

| From                   | To                     | Code                           |
|------------------------|------------------------|--------------------------------|
| <code>cstring16</code> | <code>string16</code>  | <code>string16(c)</code>       |
| <code>string16</code>  | <code>[]u16</code>     | <code>transmute([]u16)s</code> |
| <code>[]u16</code>     | <code>string16</code>  | <code>string16(x)</code>       |
| <code>cstring16</code> | <code>[^]u16</code>    | <code>([^]u16)(c)</code>       |
| <code>[^]u16</code>    | <code>cstring16</code> | <code>cstring16(p)</code>      |

Converting between UTF-8 and UTF-16 strings at runtime requires an explicit conversion, e.g. with the procedures in `core:unicode/utf16` OR `core:sys/windows` (`utf8_to_wstring`, `wstring_to_utf8`).

### string type conversions

Here is a short list of possible type conversions - including whether they *copy* or *alias*. This is important to understand since **Odin** gives you the possibility to keep *allocations* to a low degree.

If some cases are missing please let us know.

Legend:

- **copy** = get a freshly allocated copy of the 'from' data
- **alias** = reuse the 'from' data, without allocation
- **stream** = get individual values from the string, without allocation
- **st** = the input string

#### From string to X

| To      | Action | Code                                              |
|---------|--------|---------------------------------------------------|
| []u8    | alias  | <code>transmute([]u8)st</code>                    |
| string  | copy   | <code>strings.clone(st)</code>                    |
| cstring | copy   | <code>strings.clone_to_cstring(st)</code>         |
| cstring | alias  | <code>strings.unsafe_string_to_cstring(st)</code> |
| []rune  | stream | <code>for rune in st { ... }</code>               |
| []rune  | copy   | <code>utf8.string_to_runes(st)</code>             |
| [^]u8   | alias  | <code>raw_data(st)</code>                         |

#### From cstring to X

| To     | Action | Code                            |
|--------|--------|---------------------------------|
| string | alias  | <code>string(st)</code>         |
| [^]u8  | alias  | <code>transmute([_]u8)st</code> |

#### From a string literal to X

| To      | Action | Code                                                          |
|---------|--------|---------------------------------------------------------------|
| string  | alias  | <code>string(st)</code> OR <code>newstr: string = st</code>   |
| cstring | alias  | <code>cstring(st)</code> OR <code>newstr: cstring = st</code> |

#### From []u8 to X

| To     | Action | Code                                           |
|--------|--------|------------------------------------------------|
| string | alias  | <code>transmute(string)st</code>               |
| string | alias  | <code>string(st)</code> unless a slice literal |
| [^]u8  | alias  | <code>raw_data(st)</code>                      |

#### From []rune to string

| Action | Code                                  |
|--------|---------------------------------------|
| copy   | <code>utf8.runes_to_string(st)</code> |

#### From [^]u8 to cstring

| Action | Code                     |
|--------|--------------------------|
| alias  | <code>cstring(st)</code> |

## From `[^]u8` and `length int` to `string`

| Action | Code                                              |
|--------|---------------------------------------------------|
| alias  | <code>strings.string_from_ptr(ptr, length)</code> |

## Operators

Operators combine operands into expressions. For binary operations, operand types must be identical or implicitly convertible unless the operation involves shifts or untyped constants.

### Arithmetic operators

Unary:

```

+ is 0 + x
- negation is 0 - x
~ bitwise complement is m ~ x where m = "all bits set to 1" for unsigned x
 and m = -1 for signed x

```

Binary:

```

+ sum integers, enums, floats, complex values, quaternions, arrays of numeric types, matrices,
constant strings
- subtraction integers, enums, floats, complex values, quaternions, arrays of numeric types, matrices
* multiplication integers, floats, complex values, quaternions, arrays of numeric types, matrices
/ division integers, floats, complex values, quaternions, arrays of numeric types
% modulo (truncated) integers
%% remainder (floored) integers

| bitwise or integers, enums
~ bitwise xor integers, enums
& bitwise and integers, enums
&~ bitwise and-not integers, enums
<< left shift integer << integer >= 0
>> right shift integer >> integer >= 0

```

Except for shift operations, if one operand is an untyped constant and the other operand is not, the constant is implicitly converted to the type of the other operand (if possible).

The right operand in a shift expression must have an unsigned integer type or be an untyped constant representable by a typed unsigned integer. If the left operand of a non-constant shift expression is an untyped constant, it is first implicitly converted to the type it would assume if the shift expression were replaced solely by the left operand alone (with type inference and hinting rules applied).

### Comparison operators

```

== equal
!= not equal
< less
<= less or equal
> greater
>= greater or equal
&& short-circuiting logical and
|| short-circuiting logical or

```

In any comparison, the first operand must be assignable to the type of the second, or vice versa.

The equality operators `==` and `!=` apply to operands that are *comparable*. The ordering operators `<`, `<=`, `>`, and `>=` apply to operands that are *ordered*. These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable.
- Integer values are comparable and ordered.
- Floating-point values are comparable and ordered, defined by the IEEE-754 standard.
- Complex values are comparable.
- Quaternion values are comparable.
- Rune values are comparable and ordered.
- String values are comparable and ordered, lexically byte-wise.
- Matrix values are comparable.
- Pointer values are comparable and ordered.
- Multi-pointer values are comparable and ordered.
- SOA pointer values are comparable.
- Enum values are comparable and ordered.

- Bit set values are comparable.
- Struct values are comparable if all their fields are comparable.
- Union values are comparable if all their variants are comparable.
- Array and enumerated array values are comparable if values of the element type are comparable.
- `typeid` is comparable.
- SIMD vectors are comparable.

Bit set values use different logic compared to integers when comparison operators are used: please see the section on [bit sets](#).

## Logical operators

Logical operators apply to boolean values. The right operand is evaluated conditionally

```
&& conditional AND a && b is "b if a else false"
|| conditional OR a || b is "true if a else b"
! NOT !a is "not a"
```

## Compound binary operator and assign

Like many other languages, there is a shorthand for performing a binary operation and assigning the result to the first operand e.g. `x = x + 5`. All arithmetic and logical binary operators have this shorthand

```
+= sum and assign a += b is a = a + b
-= subtraction and assign a -= b is a = a - b
*= multiplication and assign a *= b is a = a * b
/= division and assign a /= b is a = a / b
%= modulo (truncated) and assign a %= b is a = a % b
%= remainder (floored) and assign a %= b is a = a % b

|= bitwise or and assign a |= b is a = a | b
^= bitwise xor and assign a ^= b is a = a ^ b
&= bitwise and and assign a &= b is a = a & b
&~= bitwise and-not and assign a &~= b is a = a &~ b
<<= left shift and assign a <<= b is a = a << b
>>= right shift and assign a >>= b is a = a >> b

&&= conditional AND and assign a &&= b is a = a && b
||= conditional OR and assign a ||= b is a = a || b
```

## Address operator

For an operand `x` of type `τ`, the address operation `&x` generates a pointer of `^τ` to `x`. The operand must be *addressable*, meaning that either a variable, pointer indirection, or slice/dynamic array indexing operator; or a field selector of an addressable struct operand; or an array index operation of an addressable array; or a type assertion of an addressable union or `any`; or a compound literal value.

For an operand `x` of pointer type `^τ`, the pointer indirection `x^` denotes the variable of type `τ` pointed to by `x`. If `x` is an invalid address, such as `nil`, an attempt to evaluate `x^` will result in platform specific behaviour - on most platforms this will be a segmentation fault.

```
&x
&a[foo(123)]
&Foo{1, 2}

p^
pproc(a)^

x: ^int = nil
x^ // causes a runtime panic
```

## Ternary operators

```
x if cond else y // ternary runtime conditional expression
x when cond else y // ternary compile-time conditional expression
cond ? x : y // ternary runtime conditional expression, equivalent to "x if cond else y"
```

## Other operators

- `or_else`
  - see section on [or\\_else](#)
- `or_return`

- see section on [or\\_return](#)
- `or_continue`
  - see section on [or\\_continue](#)
- `or_break`
  - see section on [or\\_break](#)
- `in` - set membership (`e in A`, `A` contains element `e`)
  - Used for `bit_set` types and `map` types
- `not_in` - not set membership (`e not_in A`, `A` does not contain `e`)
  - Used for `bit_set` types and `map` types
- `**` - prefix unary operator which expands a struct or array value into multiple values (`**x` is equivalent to `expand_values(x)`)
  - see section on [expand\\_values](#)
- `..=` - inclusive range
- `..<` - half open range

The range operations `..=` and `..<` are only possible within certain contexts:

```
for x in a..<b {}
for x in a..=b {}

switch x {
case a..<b:
case c..=d:
}

bit_set[a..<b]
bit_set[a..=b]

foo := [?]int{0..=3 = 1} // initialises as: [1, 1, 1, 1]
bar := [?]int{0 = 0, 1..<3 = 1} // initialises as: [0, 1, 1]
```

`in` and `not_in` cannot be used directly as a `for` loop condition, as that would be ambiguous with a range-based loop; wrap the condition in parentheses instead:

```
for x in y {} // range loop
for (x in y) {} // condition-only for-loop (while-loop in some other languages)
```

## Operator precedence

Unary operators have the highest precedence.

There are seven precedence levels for binary (and ternary) operators.

| Precedence | Operator                                                                                                                                        |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 7          | <code>*</code> <code>/</code> <code>%</code> <code>%%</code> <code>&amp;</code> <code>&amp;-</code> <code>&lt;&lt;</code> <code>&gt;&gt;</code> |
| 6          | <code>+</code> <code>-</code> <code> </code> <code>~</code> <code>in</code> <code>not_in</code>                                                 |
| 5          | <code>==</code> <code>!=</code> <code>&lt;</code> <code>&gt;</code> <code>&lt;=</code> <code>&gt;=</code>                                       |
| 4          | <code>&amp;&amp;</code>                                                                                                                         |
| 3          | <code>  </code>                                                                                                                                 |
| 2          | <code>..=</code> <code>..<code>&lt;</code></code>                                                                                               |
| 1          | <code>or_else</code> <code>?</code> <code>if</code> <code>when</code>                                                                           |

Binary operators of the same precedence associate from left to right. For instance `x / y * z` is the same as `(x / y) * z`.

## Integer operators

For two integers values `x` and `y`, the integer quotient `q = x/y` and remainder `r = x%y` satisfies the following relationships:

```
x = q*y + r and |r| < |y|
```

with `x/y` truncated towards zero ([truncated division](#)).

For two integers values `x` and `y`, the integer quotient `q = x/y` and remainder `r = x%%y` satisfies the following relationships:

```
r = x - y*floor(x/y)
```

The exception to these rules is when the dividend  $x$  is the most negative value for the integer type of  $x$ , and the quotient  $q = x/-1$  is equal to  $x$  (and  $r \text{ or } m = 0$ ) due to two's complement integer overflow.

If the divisor is a constant, it must not be zero. If the divisor is zero at runtime, a runtime panic occurs.

The shift operators shift the left operand by the shift count specified by the right operand, which must be non-negative. The shift operators implement arithmetic shifts if the left operand is a signed integer and logical shifts if the left operand is an unsigned integer. There is not an upper limit on the shift count. Shifts behave as if the left operand is shifted  $n$  times by 1 for a shift count of  $n$ . Therefore,  $x \ll 1$  is the same as  $x * 2$  and  $x \gg 1$  is the same as  $x / 2$  but truncated towards negative infinity.

```
// These are equivalent:
x << y
x << y if y < 8*size_of(x) else 0

x >> y
x >> y if y < 8*size_of(x) else 0
```

## Integer overflow

For unsigned integers, the operations  $+$ ,  $-$ ,  $*$ , and  $\ll$  are computed modulo  $2^n$ , where  $n$  is the bit width of the unsigned integer's type. In a sense, these unsigned integer operations discard the high bits upon overflow, and programs may rely on "wrap around".

For signed integers, the operations  $+$ ,  $-$ ,  $*$ ,  $/$ , and  $\ll$  may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation. Overflow **does not** cause a runtime panic. A compiler may not optimize code under the assumption that overflow does not occur. For instance,  $x < x+1$  may not be assumed to be always true.

## Floating-point operators

For floating-point, complex numbers, quaternions, and other floating-point embedded types:

- $+x$  is the same as  $x$
- $-x$  is the negation of  $x$

The result of a floating-point related division by zero is not specified beyond the IEEE-754 standard; a runtime panic will occur.

An implementation may combine multiple floating-point operations into a single fused operation, and produce a result that differs from the value obtained by executing and rounding the instructions individually.

## Advanced types

### Type alias

You can alias a named type with another name:

```
My_Int :: int
#assert(My_Int == int)
```

### Distinct types

A distinct type allows for the creation of a new type with the same underlying semantics.

```
My_Int :: distinct int
#assert(My_Int != int)
```

Aggregate types (struct, enum, union) will always be distinct even when named.

```
Foo :: struct {}
#assert(Foo != struct{})
```

### Fixed arrays

An array is a simplified fixed length container. Each element in an array has the same type. An array's index can be any integer, character, or enumeration type.

An array can be constructed like the following:

```
x := [5]int{1, 2, 3, 4, 5}
for i in 0..=4 {
```

```
 fmt.println(x[i])
}
```

Fixed arrays are equivalent to a struct with a field for each element. They are just a number of values in a row in memory.

The notation `x[i]` is used to access the *i*-th element of `x`; and 0-index based (like C).

The above array can also be constructed with a question mark (?) to automatically infer its length:

```
x := [?]int{1, 2, 3, 4, 5}
```

Construct an array with designated initializers:

```
favorite_animals := [?]string{
 // Assign by index
 0 = "Raven",
 1 = "Zebra",
 2 = "Spider",
 // Assign by range of indices
 3..5 = "Frog",
 6..<8 = "Cat"
}
```

The built-in `len` proc returns the array's length.

```
x: [5]int
#assert(len(x) == 5)
```

Array access is bounds checked by default, both at compile-time (with constant indices) and at runtime. This can be disabled and enabled at a per block level with the `#no_bounds_check` and `#bounds_check` directives, respectively:

```
#no_bounds_check {
 x[n] = 123 // n could be in or out of range of valid indices
}
```

`#no_bounds_check` can be used to improve performance when the bounds are known to not exceed.

## Array programming

Odin's fixed length arrays support [array programming](#).

Example:

```
Vector3 :: [3]f32
a := Vector3{1, 4, 9}
b := Vector3{2, 4, 8}
c := a + b // {3, 8, 17}
d := a * b // {2, 16, 72}
e := c != d // true
```

## Note

Odin also supports [matrix types](#).

## Swizzle operations

```
a := [3]f32{10, 20, 30}
b := swizzle(a, 2, 1, 0)
assert(b == [3]f32{30, 20, 10})

c := swizzle(a, 0, 0)
assert(c == [2]f32{10, 10})
assert(c == 10) // assert all elements == 10
```

Builtin implicit swizzle fields are available on any array with length  $\leq 4$  as `xyzw` and `rgba`.

```
Vector3 :: distinct [3]f32
a := Vector3{1, 2, 3}
b := Vector3{5, 6, 7}
c := (a * b)/2 + 1
d := c.x + c.y + c.z
fmt.printf("%.1f\n", d) // 22.0

cross :: proc(a, b: Vector3) -> Vector3 {
 i := swizzle(a, 1, 2, 0) * swizzle(b, 2, 0, 1)
 j := swizzle(a, 2, 0, 1) * swizzle(b, 1, 2, 0)
 return i - j
}
```

```

}

cross_shorter :: proc(a, b: Vector3) -> Vector3 {
 i := a.yzx * b.zxy
 j := a.zxy * b.yzx
 return i - j
}

blah :: proc(a: Vector3) -> f32 {
 return a.x + a.y + a.z
}

x := cross(a, b)
fmt.println(x)
fmt.println(blah(x))

```

## Array casts

A fixed length array can be converted to another fixed length array of the same length if its element type can be converted. The conversion is applied to each element, and nested arrays of the same depth are converted too:

```

a := [4]u32{1, 2, 3, 4}
b := cast([4]f32)a // [1.0, 2.0, 3.0, 4.0]
c := ([4]f64)(b) // [1.0, 2.0, 3.0, 4.0]

m := [2][2]i32{{1, 2}, {3, 4}}
n := cast([2][2]f32)m // [[1.0, 2.0], [3.0, 4.0]]

```

## Slices

Slices look similar to arrays however, their length is not known at compile time. The type `[]T` is a slice with elements of type `T`. In practice, slices are much more common than arrays.

A slice is formed by specifying two indices, a low and high bound, separated by a colon:

```
a[low : high]
```

This selects a half-open range which includes the lower element, but excludes the higher element.

```

fibonaccis := [6]int{0, 1, 1, 2, 3, 5}
s: []int = fibonaccis[1:4] // creates a slice which includes elements 1 through 3
fmt.println(s) // 1, 1, 2

```

Slices are like references to arrays; they do not store any data, rather they describe a section, or slice, of underlying data.

Internally, a slice stores a pointer to the data and an integer to store the length of the slice.

The built-in `len` proc returns the slice's length.

```

x: []int = ...
length_of_x := len(x)

```

## Slice literals

A slice literal is like an array literal without the length. This is an array literal:

```
[3]int{1, 6, 3}
```

This is a slice literal which creates the same array as above, and then creates a slice that references it:

```
[]int{1, 6, 3}
```

## Slice shorthand

For the array:

```
a: [6]int
```

these slice expressions are equivalent:

```

a[0:6]
a[:6]
a[0:]
a[:]

```

When grabbing a chunk of a slice:

```
a[offset:offset+length]
```

can also be written:

```
a[offset:][:length]
```

## Nil slices

The zero value of a slice is `nil`. A nil slice has a length of 0 and does not point to any underlying memory. Slices can be compared against `nil` and nothing else.

```
s: []int
if s == nil {
 fmt.Println("s is nil!")
}
```

## Sort slices

A slice literal can be sorted in ascending order as follows:

```
s := []int{1, 6, 3, 5, 7, 3, 0}
slice.sort(s)
```

or in descending order

```
r := []int{1, 6, 3, 5, 7, 3, 0}
slice.reverse_sort(r)
```

## Dynamic arrays

Dynamic arrays are similar to slices, but their lengths may change during runtime. Dynamic arrays are resizable and they are allocated using the current `context`'s allocator.

```
x: [dynamic]int
```

Along with the built-in proc `len`, dynamic arrays also have `cap` which can be used to determine the dynamic array's current underlying capacity.

## Appending to a dynamic array

It is common to append new elements to a dynamic array; this can be done using the built-in `append` proc.

```
x: [dynamic]int
append(&x, 123)
append(&x, 4, 1, 74, 3) // append multiple values at once

y: [dynamic]int
append(&y, ..x[:]) // append a slice
```

## Inject / assign to a dynamic array

Injecting into a specific index can be done with `inject_at`. It will move other elements upwards when inserted below other elements.

Assigning a value at a specific index can be done with `assign_at`. It's the same as doing `x[index] = 10`.

Both of these procedures will resize the dynamic array `len` to the wanted index. This can be seen in the example below.

```
x := make([dynamic]int, 0, 16)
inject_at(&x, 0, 10)
inject_at(&x, 3, 10) // resizes till length
fmt.Println(x[:], len(x), cap(x)) // [10, 0, 0, 10] 4 16
assign_at(&x, 3, 20)
assign_at(&x, 4, 30)
fmt.Println(x[:], len(x), cap(x)) // [10, 0, 0, 20, 30] 5 16
assign_at(&x, 5, 40, 50, 60)
fmt.Println(x[:], len(x), cap(x)) // [10, 0, 0, 20, 30, 40, 50, 60] 8 16
```

## Removing from a dynamic array

Removing from a dynamic array can be done in several ways using the built-in procedures:

- `pop` pops the last element of the array

- `unordered_remove` removes an element at a specific index. *Unordered* means it is  $O(1)$ , since it swaps the last element to the removed location - making the array be *sorted* differently.
- `ordered_remove` removes an element at a specific index. *Ordered* means it will move all elements after the index downwards with a *copy* - ensuring elements remain in the same order.

```
x: [dynamic]int
append(&x, 1, 2, 3, 4, 5) // [1, 2, 3, 4, 5]
pop(&x) // [1, 2, 3, 4]
ordered_remove(&x, 0) // [2, 3, 4]
unordered_remove(&x, 0) // [4, 3]
```

Other variants can be found in the [built-in procedures](#) documentation.

### Slice & sort a dynamic array

Although dynamic arrays and slices are different concepts, dynamic arrays can be 'sliced' and sorted as follows:

```
s: [dynamic]int
append(&s, 1, 6, 3, 5, 7, 3, 0) // [1, 6, 3, 5, 7, 3, 0]
slice.sort(s[:]) // [0, 1, 3, 3, 5, 6, 7]
```

### Making and deleting slices and dynamic arrays

Slices and dynamic arrays can be explicitly allocated with the built-in `make` proc.

```
a := make([int, 6) // len(a) == 6
b := make([dynamic]int, 6) // len(b) == 6, cap(b) == 6
c := make([dynamic]int, 0, 6) // len(c) == 0, cap(c) == 6
d := [int]{1, 2, 3} // a slice literal, for comparison

// with an explicit allocator:
e := make([int, 6, context.allocator)
f := make([dynamic]int, 0, 6, context.allocator)
```

Slices and dynamic arrays can be deleted with the built-in `delete` proc.

```
delete(a)
delete(b)
delete(c)
// delete(d) // no need to clean up slice literals
delete(e) // slices are always deleted from context.allocator
delete(f) // dynamic arrays remember their allocator
```

### Note

There is no automatic memory management in Odin.

### Clearing a dynamic array

Instead of deleting the array you often want to simply `clear` the dynamic array. This will set the length `len()` to be 0, while the capacity `cap` remains the same.

```
x: [dynamic]int
append(&x, 1, 2, 3, 4, 5) // [1, 2, 3, 4, 5]
fmt.println(len(x)) // 5
clear(&x) // []
fmt.println(len(x)) // 0
```

### Resize / reserve with a dynamic array

Often enough we also want to resize or reserve a specific amount for a dynamic array. It's important to understand the difference between the two operations.

- `resize` will try to resize memory of a passed dynamic array to the requested element count (setting the `len`, and possibly `cap`).
- `reserve` will try to reserve memory of a passed dynamic array to the requested element count (setting the `cap`).
- `shrink` will shrink the capacity of a dynamic array down to the current length, or the given capacity.

```
x: [dynamic]int
fmt.println(len(x), cap(x)) // 0 0
append(&x, 1, 2, 3) // [1, 2, 3]
fmt.println(len(x), cap(x)) // 3 8
```

```

resize(&x, 5)
fmt.println(x[:]) // [1, 2, 3, 0, 0] other values are zero'd memory
fmt.println(len(x), cap(x)) // 5 8
reserve(&x, 32)
fmt.println(len(x), cap(x)) // 5 32
shrink(&x)
fmt.println(len(x), cap(x)) // 5 5

```

### [dynamic; N]T fixed capacity dynamic array

Odin supports a fixed capacity dynamic array [dynamic; N]T. It implements most dynamic array procedures while being able to remain on the stack.

Short Example:

```

x: [dynamic; 8]int
fmt.println(len(x), cap(x)) // 0 8
append(&x, 1, 2, 3)
fmt.println(len(x), cap(x)) // 3 8
fmt.println(x[:]) // [1, 2, 3]

```

A fixed capacity dynamic array is a value type: it stores its *n* elements inline along with its length, and it never allocates, meaning it has no allocator and never needs to be deleted. Its capacity is always *n*.

Because it cannot grow, `append` returns the number of elements actually appended, which is 0 if there is no room:

```

x: [dynamic; 4]int
n := append(&x, 1, 2, 3) // n == 3
n = append(&x, 4, 5) // n == 1, only `4` fits
n = append(&x, 6) // n == 0, the array is full
fmt.println(x[:]) // [1, 2, 3, 4]

```

The following built-in procedures support fixed capacity dynamic arrays: `append`, `non_zero_append`, `inject_at`, `assign_at`, `pop`, `pop_safe`, `pop_front`, `pop_front_safe`, `unordered_remove`, `ordered_remove`, `remove_range`, `clear`, `resize`, and `non_zero_resize`. They can also be sliced, iterated with `for` loops, and initialized with compound literals:

```

y := [dynamic; 8]int{1, 2, 3}
fmt.println(len(y), cap(y)) // 3 8

```

### Note

This replaces the core library container dynamic array `Small_Array(N, T)`.

## Enumerations

Enumeration types define a new type whose values consist of the ones specified. The values are ordered, for example:

```

Direction :: enum{North, East, South, West}

```

The following holds:

```

int(Direction.North) == 0
int(Direction.East) == 1
int(Direction.South) == 2
int(Direction.West) == 3

```

Enum fields can be assigned an explicit value:

```

Foo :: enum {
 A,
 B = 4, // Holes are valid
 C = 7,
 D = 1337,
}

```

If an enumeration requires a specific size, a backing integer type can be specified. By default, `int` is used as the backing type for an enumeration.

```

Foo :: enum u8 {A, B, C} // Foo will only be 8 bits

```

## Implicit selector expression

An *implicit selector expression* is an abbreviated way to access a member of an enumeration, in a context where type inference can determine the implied type. It has the following form:

```
.member_name
```

For example:

```
Foo :: enum{A, B, C}
f: Foo
f = Foo.A
f = .A

switch f {
case .A:
 fmt.println("foo")
case .B:
 fmt.println("bar")
case .C:
 fmt.println("baz")
}
```

A [using statement](#) can also be used with an enumeration to bring the fields into the current scope, but this requires `#+feature using-stmt` in the file:

```
#+feature using-stmt
package main

main :: proc() {
 Foo :: enum {A, B, C}
 using Foo
 a := A
}
```

### Note

Implicit selector expression is preferred to [using](#) an enumeration as `using` does pollute the current scope.

## Iterating an enumeration

Enums can be trivially `for` looped in `odin`. This way we can loop through the entire `enum` and do things like printing or inserting into an *Enumerated Array*.

```
Direction :: enum{North, East, South, West}

for direction, index in Direction {
 fmt.println(index, direction)
 // 0 North
 // 1 East
 // 2 South
 // 3 West
}
```

## Enumerated array

Enumerated arrays allow an `enum` to be used as the index type of a fixed array.

We'll extend the `Direction` enum used previously to add direction vectors.

```
Direction :: enum{North, East, South, West}

Direction_Vectors :: [Direction][2]int {
 .North = { 0, -1 },
 .East = { +1, 0 },
 .South = { 0, +1 },
 .West = { -1, 0 },
}

assert(Direction_Vectors[.North] == { 0, -1 })
assert(Direction_Vectors[.East] == { 1, 0 })
assert(Direction_Vectors[cast(Direction) 2] == { 0, 1 })
```

The `#partial` directive can be used to initialize an enumerated array *partially*.

The `#sparse` directive can be used to initialize an enumerated array with an `enum` which does not have contiguous values.

```
arr: [enum {A, B, C}]int
arr = #partial { // without partial the compiler would complain
 .A = 42,
```

```
}
fmt.println(arr) // [.A = 42, .B = 0, .C = 0]
```

## Bit sets

The `bit_set` type models the mathematical notion of a set. A `bit_set`'s element type can be either an enumeration or a range:

```
Direction :: enum{North, East, South, West}
Direction_Set :: bit_set[Direction]
Char_Set :: bit_set['A'..'Z']
Number_Set :: bit_set[0..<10] // bit_set[0..=9]
```

Bit sets are implemented as bit vectors internally for high performance. The zero value of a bit set is either `nil` or `{}`.

```
x: Char_Set
x = {'A', 'B', 'Y'}
y: Direction_Set
y = {.North, .West}
```

Bit sets support the following operations:

- `A + B` - union of two sets (equivalent to `A | B`)
- `A - B` - difference of two sets (A without B's elements) (equivalent to `A &~ B`)
- `A & B` - intersection of two sets
- `A | B` - union of two sets (equivalent to `A + B`)
- `A &~ B` - difference of two sets (A without B's elements) (equivalent to `A - B`)
- `A ~ B` - symmetric difference (Elements that are in A and B but not both)
- `A == B` - set equality
- `A != B` - set inequality
- `A <= B` - subset relation (A is a subset of B or equal to B)
- `A < B` - strict subset relation (A is a proper subset of B)
- `A >= B` - superset relation (A is a superset of B or equal to B)
- `A > B` - strict superset relation (A is a proper superset of B)
- `e in A` - set membership (A contains element e)
- `e not_in A` - not set membership (A does not contain element e)

Bit sets are often used to denote flags. This is much cleaner than defining integer constants that need to be bitwise or-ed together.

If a bit set requires a specific size, the underlying integer type can be specified:

```
Char_Set :: bit_set['A'..'Z'; u64]
#assert(size_of(Char_Set) == size_of(u64))
```

The underlying type may also be a fixed length array of integers. Integer backed bit sets are limited to 128 bits; an array backing allows for bit sets with more elements:

```
Big :: enum {A = 0, B = 100, C = 200}
Big_Set :: bit_set[Big; [4]u64] // 256 bits
#assert(size_of(Big_Set) == 32)

s := Big_Set{.A, .C}
assert(.C in s)
assert(card(s) == 2)
```

To get the number of elements set, its cardinality, of a `bit_set`, use the built-in `card` procedure:

```
x: Direction_Set
x = {.North, .West}
count := card(x)
assert(count == 2)
```

## Pointers

Odin has pointers. A pointer is a memory address of a value. The type `^τ` is a pointer to a `τ` value. Its zero value is `nil`.

```
p: ^int
```

The `&` operator takes the address of its operand (if possible):

```
i := 123
p := &i
```

The `^` operator dereferences the pointer's underlying value:

```
fmt.Println(p^) // read `i` through the pointer `p`
p^ = 1337 // write `i` through the pointer `p`
```

### Note

C programmers may be used to using `*` to denote pointers. In Odin, the `^` syntax is borrowed from Pascal. This is to keep the convention of the type on the left and its usage on the right:

```
p: ^int // ^ on the left
x := p^ // ^ on the right
```

### Note

Unlike C, `^T` pointers do not support pointer arithmetic. If you need a form of pointer arithmetic, please use a [multi-pointer](#) (`[^]T`) which supports indexing and slicing, or the `ptr_offset` and `ptr_sub` procedures in `base:intrinsic`s (also available in the `core:mem` package).

## Structs

A struct is a record type in Odin. It is a collection of fields. Struct fields are accessed by using a dot:

```
Vector2 :: struct {
 x: f32,
 y: f32,
}
v := Vector2{1, 2}
v.x = 4
fmt.Println(v.x)
```

Struct fields can be accessed through a struct pointer:

```
v := Vector2{1, 2}
p := &v
p.x = 1335
fmt.Println(v)
```

We could write `p^.x`, however, it is nice to not have to explicitly dereference the pointer. This is very useful when refactoring code to use a pointer rather than a value, and vice versa.

### Struct literals

A struct literal can be denoted by providing the struct's type followed by `{}`. A struct literal must either provide all the arguments or none:

```
Vector3 :: struct {
 x, y, z: f32,
}
v: Vector3
v = Vector3{} // Zero value
v = Vector3{1, 4, 9}
```

You can list just a subset of the fields if you specify the field by name (the order of the named fields does not matter):

```
v := Vector3{z=1, y=2}
assert(v.x == 0)
assert(v.y == 2)
assert(v.z == 1)
```

Structs can be nested by defining a field as a struct.

```
Foo :: struct {
 a, b, c: int,
```

```

bar_1: struct {
 x, y, z: int,
},

bar_2: struct {
 x, y, z: int,
},

_: struct {
 x, y, z: int,
},
}

```

## Struct directives

Structs can be annotated with different memory layout and alignment requirements:

```

struct #align(4) {...} // align to 4 bytes
struct #raw_union {...} // all fields share the same offset (0). This is the same as C's union
struct #packed {...} // remove padding between fields
struct #min_field_align(4) {...} // all fields must have a minimum alignment of 4 bytes (equivalent to `#pragma pack(4)` in C extensions)
struct #max_field_align(4) {...} // all fields must have a maximum alignment of 4 bytes
struct #simple {...} // The struct is compared as if with `memcmp`, without needing any other special behaviour

```

## #all\_or\_none

The directive `#all_or_none` can be applied to a struct. This prevents partial initialization of the struct compound literal, by requiring it is either all or the fields are specified or none of the fields are specified.

```

Foo :: struct #all_or_none {
 a: int,
 b: int,
 c: int,
}

test :: proc() {
 // No fields set
 a := Foo{}

 // This is an error.
 b := Foo{
 a = 10,
 }

 // All fields set
 c := Foo{
 a = 10,
 b = 10,
 c = 10,
 }
}

```

## Struct field tags

Struct fields can be tagged with a string literal to attach meta-information which can be used with runtime-type information. Usually this is used to provide transactional information info on how a struct field is encoded to or decoded from another format, but you can store whatever you want within the string literal

```

User :: struct {
 flag: bool, // untagged field
 age: int "custom whatever information",
 name: string `json:"username" xml:"user-name" fmt:"q"`, // `core:reflect` layout
}

```

Within Odin's core library, the standard convention is to store a key that denotes the package and then a sub-sequence "value". For example, `json` keys are processed and used by `core:encoding/json` package, `fmt` keys are processed by `core:fmt`.

If multiple information is to be passed in the "value", usually it is specified by separating it with a common `(.)`, e.g.

```
name: string `json:"username,omitempty"`,
```

## Note

Field tags also exist for [bit\\_field](#) record types.

## Unions

A union in Odin is a discriminated union, also known as a tagged union or sum type. The zero value of a union is `nil`.

```
Value :: union {
 bool,
 i32,
 f32,
 string,
}
v: Value
v = "Hellope"

// type assert that `v` is a `string` and panic otherwise
s1 := v.(string)

// type assert but with an explicit boolean check. This will not panic
s2, ok := v.(string)
```

## Type switch statement

A type switch is a construct that allows several type assertions in series. A type switch is like a regular switch statement, but the cases are types (not values). For a union, the only case types allowed are that of the union.

```
value: Value = ...
switch v in value {
case string:
 #assert(type_of(v) == string)

case bool:
 #assert(type_of(v) == bool)

case i32, f32:
 // This case allows for multiple types, therefore we cannot know which type to use
 // `v` remains the original union value
 #assert(type_of(v) == Value)
case:
 // Default case
 // In this case, it is `nil`
}
```

## Union tags

Applying the `#no_nil` tag to a union type states it does not have a `nil` value. Unions with `#no_nil` must have at least two variants and the first variant is its default type:

```
Value :: union #no_nil {bool, string}
v: Value
_, ok := v.(bool)
assert(ok)
```

The `#shared_nil` tag normalizes each variant's `nil` value into `nil` on assignment. If you assign `nil` or [zero values](#) to a union with `#shared_nil` the union will be `nil`. Unions with `#shared_nil` require all variants to have a `nil` value.

```
Error :: union #shared_nil {
 File_Error,
 Memory_Error,
}

File_Error :: enum {
 None = 0,
 File_Not_Found,
 Cannot_Open_File,
}

Memory_Error :: enum {
 None = 0,
 Allocation_Failed,
 Resize_Failed,
}

shared_nil_example :: proc() {
 an_error: Error
 an_error = File_Error.None

 assert(an_error == nil)
}
```

Unions also have the `#align` tag, like structures:

```
union #align(4) {...} // align to 4 bytes
```

## Maps

A map maps keys to values. The zero value of a map is `nil`. A `nil` map has no keys. The built-in `make` proc returns an initialized map using the current `context`, and `delete` can be used to delete a map.

```
m := make(map[string]int)
defer delete(m)
m["Bob"] = 2
fmt.Println(m["Bob"])
```

To insert or update an element of a map:

```
m[key] = elem
```

To retrieve an element:

```
elem = m[key]
```

To remove an element:

```
delete_key(&m, key)
```

If an element of a key does not exist, the zero value of the element will be returned. Checking to see if an element exists can be done in two ways:

```
elem, ok := m[key] // `ok` is true if the element for that key exists
```

or

```
ok := key in m // `ok` is true if the element for that key exists
```

The first approach is called the "comma ok idiom".

You can also initialize maps with map literals:

```
m := map[string]int{
 "Bob" = 2,
 "Chloe" = 5,
}
```

### Note

To enable compound literals for `mapS`, `#+feature dynamic-literals` must be enabled per file. This is because dynamic literals will use the current `context allocator` and thus implicitly allocate. The opt-in feature is there in order for Odin to not implicitly allocate by default and not give the user any surprises.

Modifying existing map slots needs to be done in two steps. However assigning to a struct field is prohibited.

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

m := map[string]Test{
 "Bob" = { 0, 0 },
 "Chloe" = { 1, 1 },
}

value, ok := &m["Bob"]
if ok {
 value^ = { 2, 2 }
}

fmt.Println(m["Bob"]) // { 2, 2 }
m["Bob"] = { 3, 3 }
fmt.Println(m["Bob"]) // { 3, 3 }
m["Chloe"].x = 0 // PROHIBITED
```

## Map container calls

The built-in map also supports all the standard container calls that can be found with the `dynamic array`. Short:

- `len(some_map)` returns the amount of slots used up
- `cap(some_map)` returns the capacity of the map - the map will reallocate when exceeded
- `clear(&some_map)` clears the entire map - dynamically allocated content needs to be freed manually
- `reserve(&some_map, capacity)` reserves the requested element count
- `shrink(&some_map)` shrink the capacity down to the current length

## Bit fields

A `bit_field` is a record type in Odin that is akin to a bit-packed struct.

### Note

`bit_field` is **not** equivalent to `bit_set` as it has different semantics and use cases.

`bit_field` fields are accessed by using a dot:

```
Foo :: bit_field u16 { // backing type must be an integer or array of integers
 x: i32 | 3, // signed integers will be signed extended on use
 y: u16 | 2 + 3, // general expressions
 z: My_Enum | foo.SOME_CONSTANT, // ability to define the bit-width elsewhere
 w: bool | 2 when foo.SOME_CONSTANT > 10 else 1,
}

v := Foo{}
v.x = 3 // truncates the value to fit into 3 bits
fmt.println(v.x) // accessing will convert `v.x` to an `i32` and do an appropriate sign extension
```

A `bit_field` is different from a `struct` in that you must specify the backing type. This backing type must be an integer or a fixed-length array of integers. This is useful if there needs to be a specific alignment or access pattern for the record.

```
Foo :: bit_field u32 {...}
Foo :: bit_field [4]u8 {...}
```

Notes:

- If *all* of the fields in a `bit_field` are 1-bit in size and are a boolean, please consider using a `bit_set` instead.
- Odin's `bit_field` and C's bit-fields might not be compatible
  - ▶ Odin's `bit_fields` have a well defined layout (Least-Significant-Bit)
  - ▶ C's bit fields on `structs` are undefined and are not portable across targets and compilers

## Procedure type

A procedure type is internally a pointer to a procedure in memory. `nil` is the zero value of a procedure type. Examples:

```
proc(x: int) -> bool
proc(c: proc(x: int) -> bool) -> (i32, f32)
```

Or you can assign them to a variable:

```
Callback :: proc() -> int
a: Callback // nil
assert(a == nil)
a = proc() -> int { return 0 }
fmt.println(a()) // 0
a = proc() -> int { return 100 }
fmt.println(a()) // 100
```

## Calling conventions

Odin supports the following calling conventions:

- **odin** - default convention used for an Odin **proc**. It passes all parameters larger than 16 bytes by reference and passes an implicit `context` pointer on each call. (**Note:** This is subject to change)
- **contextless** - This is the same as **odin** but without the implicit `context` pointer.
- **stdcall** or **std** -- This is the `stdcall` convention as specified by Microsoft.
- **cdecl** or **c** -- This is the default calling convention generated of a procedure in C.
- **fastcall** or **fast** - This is a compiler dependent calling convention.
- **none** - This is a compiler dependent calling convention which will do nothing to parameters.

- **system** - **stdcall** on Windows and **c** on all other platforms. This is the convention generally used for operating system APIs, e.g. throughout `core:sys/windows`.
- **naked** - The procedure has no prologue or epilogue generated by the compiler. This is intended for procedures whose body is entirely written in inline assembly.
- **win64** - The Microsoft x64 calling convention, regardless of the target operating system (amd64 only).
- **sysv** - The System V AMD64 ABI calling convention, regardless of the target operating system (amd64 only).
- **preserve/none** - The callee does not preserve any general purpose registers. This minimizes the cost of calls (especially `#must_tail` calls), e.g. for the dispatch loop of an interpreter.
- **preserve/most** - The callee preserves most registers, minimizing the cost for the caller. This is useful for rarely called procedures.
- **preserve/all** - Like **preserve/most** but the callee also preserves the floating point and vector registers.

Most calling conventions exist only to interface with `foreign` code.

The default calling convention is **odin**, unless it is within a `foreign` block, where it is then **cdecl**.

A procedure type with a different calling convention can be declared like the following:

```
proc "c" (n: i32, data: rawptr)
proc "contextless" (s: []int)
```

Procedure types are only compatible with the procedures that have the same calling convention and parameter types.

When binding to C libraries you'll often end up using `proc "c"` and also set the current context. For this you'll need to [explicitly set the context](#).

### typeid type

A `typeid` is a unique identifier for an Odin type. This construct is used by the `any` type to denote what the underlying data's type is.

```
a := typeid_of(bool)
i: int = 123
b := typeid_of(type_of(i))
```

A `typeid` can be mapped to relevant type information which can be used in applications such as printing types and editing data:

```
import "base:runtime"

main :: proc() {
 u := u8(123)
 id := typeid_of(type_of(u))
 info: ^runtime.Type_Info
 info = type_info_of(id)
}
```

### any Type

An `any` type can reference any data type. Internally it contains a pointer to the underlying data and its relevant `typeid`. This is a very useful construct in order to have a runtime type safe printing procedure.

#### Note

The `any` value is only valid for as long as the underlying data is still valid. Passing a literal to an `any` will allocate the literal in the current stack frame.

#### Note

It is highly recommended that you **do not** use this unless you know what you are doing. Its primary use is for printing procedures.

### Multi-pointers

Multi-Pointers in Odin are a way to describe `foreign` (C-like) pointers which act like arrays (pointers that map to multiple items). The type `[^]T` is a multi-pointer to `T` value(s). Its zero value is `nil`.

```
p: [^]int
```

What multi-pointers support:

- Index (without any bounds checking)
- Slicing (bounds checking on if both the low and high operands are given)
- Implicit conversions between `^T` and `[^]T`
- Implicit conversion to `rawptr` (like all pointers)

What multi-pointers DO NOT SUPPORT:

- Dereferencing (which makes it closer to a slim-slice than a pointer)

The main purpose of this type is to aid with `foreign` code and act as a way to auto-document functionality and allow for easier transition to Odin code, especially converting pointers into slices.

The following are the rules for indexing and slicing for multi-pointers, and what type they produce depending on the operands given:

```
x: [^]T = ...
x[i] -> T
x[:i] -> [^]T
x[i:i] -> [^]T
x[:n] -> []T
x[i:n] -> []T
```

Interacting with Multi-Pointers is easiest using the builtin `raw_data()` call which can return a Multi-Pointer.

```
a: [^]int
fmt.println(a) // <nil>
b := [?]int { 10, 20, 30 }
a = raw_data(b[:])
fmt.println(a, a[1], b) // 0x7FFCBE9FE688 20 [10, 20, 30]
```

## SOA data types

*Array of Structures (AoS)*, *Structure of Arrays (SoA)*, and *Array of Structures of Arrays (AoSoA)* refer to differing ways to arrange a sequence of data records in memory, with regard to interleaving. These are of interest in SIMD and SIMT programming.

### SOA struct arrays

```
Vector3 :: struct {x, y, z: f32}

N :: 2
v_aos: [N]Vector3
v_aos[0].x = 1
v_aos[0].y = 4
v_aos[0].z = 9

fmt.println(len(v_aos))
fmt.println(v_aos[0])
fmt.println(v_aos[0].x)
fmt.println(&v_aos[0].x)

v_aos[1] = {0, 3, 4}
v_aos[1].x = 2
fmt.println(v_aos[1])
fmt.println(v_aos)

v_soa: #soa[N]Vector3

v_soa[0].x = 1
v_soa[0].y = 4
v_soa[0].z = 9

// Same syntax as AOS and treat as if it was an array
fmt.println(len(v_soa))
fmt.println(v_soa[0])
fmt.println(v_soa[0].x)
fmt.println(&v_soa[0].x)
v_soa[1] = {0, 3, 4}
v_soa[1].x = 2
fmt.println(v_soa[1])

// Can use SOA syntax if necessary
v_soa.x[0] = 1
v_soa.y[0] = 4
v_soa.z[0] = 9
fmt.println(v_soa.x[0])
```

```
// Same pointer addresses with both syntaxes
assert(&v_soa[0].x == &v_soa.x[0])

// Same fmt printing
fmt.println(v_aos)
fmt.println(v_soa)
```

Works with arrays of length  $\leq 4$  which have the implicit fields `xyzw/rgba`

```
Vector3 :: distinct [3]f32

N :: 2
v_aos: [N]Vector3
v_aos[0].x = 1
v_aos[0].y = 4
v_aos[0].z = 9

v_soa: #soa[N]Vector3

v_soa[0].x = 1
v_soa[0].y = 4
v_soa[0].z = 9
```

### SOA struct slices and dynamic arrays

Fixed-length SOA types can be sliced to produce SOA slices.

```
Vector3 :: struct {x: i8, y: i16, z: f32}

N :: 3
v: #soa[N]Vector3
v[0].x = 1
v[0].y = 4
v[0].z = 9

s: #soa[]Vector3
s = v[:]
assert(len(s) == N)
fmt.println(s)
fmt.println(s[0].x)

a := s[1:2]
assert(len(a) == 1)
fmt.println(a)
```

To be complete with SOA slices, Odin also supports SOA dynamic arrays.

```
d: #soa[dynamic]Vector3

append_soa(&d, Vector3{1, 2, 3}, Vector3{4, 5, 9}, Vector3{-4, -4, 3})
fmt.println(d)
fmt.println(len(d))
fmt.println(cap(d))
fmt.println(d[:])
```

The general dynamic array built-in procedures also work with `#soa` dynamic arrays, including `make`, `delete`, `append`, `inject_at`, `pop`, `pop_safe`, `pop_front`, `pop_front_safe`, `unordered_remove`, `ordered_remove`, `clear`, `reserve`, and `resize`:

```
d: #soa[dynamic]Vector3
defer delete(d)

append(&d, Vector3{1, 2, 3}, Vector3{4, 5, 6}, Vector3{7, 8, 9})
last := pop(&d) // Vector3{7, 8, 9}
first := pop_front(&d) // Vector3{1, 2, 3}
assert(len(d) == 1)
```

### SOA pointers

Taking the address of an element of an `#soa` array, slice, or dynamic array produces an *SOA pointer*, `#soa^#soa[N]T`, `#soa^#soa[]T`, or `#soa^#soa[dynamic]T` respectively. An SOA pointer refers to a single element (a pointer to the SOA data and an index), and its fields can be accessed and modified like a normal pointer to a struct:

```
v: #soa[4]Vector3
p := &v[2] // p: #soa^#soa[4]Vector3
p.x = 123
assert(v[2].x == 123)
assert(v.x[2] == 123)
```

using can be applied to SOA pointer fields of structs, which allows the same subtyping behaviour as a normal pointer field.

### soa\_zip and soa\_unzip

SOA is not just useful for high performance scenarios but also for everyday tasks which are normally only achievable in higher level languages. `soa_zip` is a built-in procedure which allows the user to treat multiple slices as if they are part of the same data structures, utilizing the power of SOA.

```
x := []i32{1, 3, 9}
y := []f32{2, 4, 16}
z := []b32{true, false, true}

// produce an #soa slice with the normal slices passed
s := soa_zip(a=x, b=y, c=z)

// iterate over the #soa slice
for v, i in s {
 fmt.println(v, i) // exactly the same as s[i]
 // NOTE: `v` is NOT a temporary value but has a specialized addressing mode
 // which means that when accessing v.a etc, it does the correct transformation
 // internally:
 // s[i].a == s.a[i]
 fmt.println(v.a, v.b, v.c)
}
```

`soa_unzip` is a built-in procedure which allows the user to recover the slices from an `#soa` slice.

```
// Recover the slices from the #soa slice
a, b, c := soa_unzip(s)
fmt.println(a, b, c)
```

### matrix type

A `matrix` is a `mathematical type` built into Odin. It is a regular array of numbers, arranged in rows and columns.

The following represents a matrix that has 2 rows and 3 columns:

```
m: matrix[2, 3]f32

m = matrix[2, 3]f32{
 1, 9, -13,
 20, 5, -6,
}
```

Element types of integers, float, and complex numbers are supported by matrices. There is no support for booleans, quaternions, or any compound type.

Indexing a matrix can be used with the matrix indexing syntax. This mirrors other type usages: type on the left, usage on the right.

```
elem := m[1, 2] // row 1, column 2
```

Scalars act as if they are scaled identity matrices and can be assigned to matrices as them

```
b := matrix[2, 2]f32{}
f := f32(3)
b = f

fmt.println("b", b)
fmt.println("b == f", b == f)
```

Matrices support multiplication between matrices:

```
a := matrix[2, 3]f32{
 2, 3, 1,
 4, 5, 0,
}

b := matrix[3, 2]f32{
 1, 2,
 3, 4,
 5, 6,
}

fmt.println("a", a)
fmt.println("b", b)

c := a * b
#assert(type_of(c) == matrix[2, 2]f32)
fmt.tprintln("c = a * b", c)
```

Matrices support multiplication between matrices and arrays:

```
m := matrix[4, 4]f32{
 1, 2, 3, 4,
 5, 5, 4, 2,
 0, 1, 3, 0,
 0, 1, 4, 1,
}

v := [4]f32{1, 5, 4, 3}

// treating `v` as a column vector
fmt.println("m * v", m * v)

// treating `v` as a row vector
fmt.println("v * m", v * m)

// Support with non-square matrices
s := matrix[2, 4]f32{ // [4][2]f32 default layout
 2, 4, 3, 1,
 7, 8, 6, 5,
}

w := [2]f32{1, 2}
r: [4]f32 = w * s
fmt.println("r", r)
```

Component-wise operations:

```
// if the element type supports it
// Not support for '/', '%', or '%%' operations

a := matrix[2, 2]i32{
 1, 2,
 3, 4,
}

b := matrix[2, 2]i32{
 -5, 1,
 9, -7,
}

c0 := a + b
c1 := a - b
c2 := a & b
c3 := a | b
c4 := a ~ b
c5 := a &~ b

// component-wise multiplication
// since a * b would be a standard matrix multiplication
c6 := hadamard_product(a, b)

fmt.println("a + b", c0)
fmt.println("a - b", c1)
fmt.println("a & b", c2)
fmt.println("a | b", c3)
fmt.println("a ~ b", c4)
fmt.println("a &~ b", c5)
fmt.println("hadamard_product(a, b)", c6)
```

## Submatrix casting

### Submatrix casting square matrices

Casting a square matrix to another square matrix with same element type is supported.

- If the cast is to a smaller matrix type, the top-left submatrix is taken.
- If the cast is to a larger matrix type, the matrix is extended with zeros everywhere and ones in the diagonal for the unfilled elements of the extended matrix.

```
mat2 :: distinct matrix[2, 2]f32
mat4 :: distinct matrix[4, 4]f32

m2 := mat2{
 1, 3,
 2, 4,
}

m4 := mat4(m2)
assert(m4[2, 2] == 1)
assert(m4[3, 3] == 1)
fmt.printf("m2 %#v\n", m2)
fmt.println("m4", m4)
fmt.println("mat2(m4)", mat2(m4))
assert(mat2(m4) == m2)

b4 := mat4{
```

```

1, 2, 0, 0,
3, 4, 0, 0,
5, 0, 6, 0,
0, 7, 0, 8,
}
fmt.println("b4", matrix_flatten(b4))

```

### Casting non-square matrices

Casting a matrix to another matrix is allowed as long as they share the same element type and the number of elements (rows\*columns). Matrices in Odin are stored in column-major order, which means the casts will preserve this element order.

```

mat2x4 :: distinct matrix[2, 4]f32
mat4x2 :: distinct matrix[4, 2]f32

x := mat2x4{
 1, 3, 5, 7,
 2, 4, 6, 8,
}

y := mat4x2(x)
fmt.println("x", x)
fmt.println("y", y)

```

### Technical information of `matrix` types

The default internal representation of a matrix in Odin is stored in column-major format e.g. `matrix[2, 3]f32` is internally `[3][2]f32` (with a different alignment requirement).

Column-major is used in order to utilize (SIMD) vector instructions effectively on modern hardware, if possible.

If a row-major storage format is required, the `#row_major` directive can be used. e.g. `#row_major matrix[2, 3]f32` is internally `[2][3]f32` (with a different alignment requirement).

Unlike normal arrays, matrices try to maximize alignment to allow for the (SIMD) vectorization properties whilst keeping zero padding (either between columns (assuming default layout) or at the end of the type).

Zero padding is a compromise for use with third-party libraries, instead of optimizing for performance.

Padding between columns (assuming default layout) was not taken even if that would have allowed each column to be loaded individually into a SIMD register with the correct alignment properties.

Currently, matrices are limited to a maximum of 64 elements (rows\*columns), and a minimum of 1 element.

This is because matrices are stored as values (not a reference type), and thus operations on them will be stored on the stack. Restricting the maximum element count minimizes the possibility of stack overflows.

Built-in Procedures (Compiler Level) in `base:intrinsics` and `core:math/linalg`:

- `transpose(m)` transposes a matrix
- `outer_product(a, b)` takes two array-like data types and returns the outer product of the values in a matrix
- `hadamard_product(a, b)` component-wise multiplication of two matrices of the same type
- `matrix_flatten(m)`
  - converts the matrix into a flatten array of elements in column-major order.
- `conj(x)`
  - conjugates the elements of a matrix for complex element types only

Procedures (Runtime Level) (all square matrix procedures) in `core:math/linalg`:

- `determinant(m)`
- `adjugate(m)`
- `inverse(m)`
- `inverse_transpose(m)`
- `hermitian_adjoint(m)`
- `matrix_trace(m)`
- `matrix_minor(m)`

### Complex numbers and quaternions

Odin has built-in [complex number](#) types (`complex32`, `complex64`, `complex128`) and [quaternion](#) types (`quaternion64`, `quaternion128`, `quaternion256`). The number in the type name is the total size in bits, e.g. a `complex64` is made of two `f32` values and a `quaternion256` is made of four `f64` values.

Values can be written using the imaginary literal suffixes `i`, `j`, and `k`, or constructed with the built-in procedures `complex` and `quaternion`:

```
c := 1 + 2i // untyped complex, defaults to complex128
f: f32 = 3
d := complex(f, 4) // complex64, from the type of `f`
e: complex64 = 3 + 4i

q := 1 + 2i + 3j + 4k // untyped quaternion, defaults to quaternion256
r := quaternion(w=1, x=2, y=3, z=4) // `quaternion` requires named arguments
s := quaternion(real=1, imag=2, jmag=3, kmag=4) // equivalent
assert(q == r && r == s)
```

They support the arithmetic operators `+`, `-`, `*`, and `/`, and are [comparable](#) with `==` and `!=`. Floats implicitly convert to complex numbers and quaternions, and complex numbers implicitly convert to quaternions.

Built-in procedures for complex numbers and quaternions:

- `real(x)` - the real part
- `imag(x)` - the `i` imaginary part
- `jmag(q)`, `kmag(q)` - the `j` and `k` imaginary parts of a quaternion
- `conj(x)` - the conjugate
- `abs(x)` - the magnitude

```
c := 3 + 4i
fmt.println(real(c), imag(c)) // 3 4
fmt.println(conj(c)) // 3-4i
fmt.println(abs(c)) // 5
```

The components of a quaternion can also be accessed as the fields `x`, `y`, `z` (the imaginary parts), and `w` (the real part). The vector part can be accessed with `xyz` as a `[3]` array of the component type:

```
q: quaternion128 = 1 + 2i + 3j + 4k
assert(q.w == 1 && q.x == 2 && q.y == 3 && q.z == 4)
v := q.xyz // [3]f32{2, 3, 4}
q.xyz = {10, 20, 30}
fmt.println(q) // 1+10i+20j+30k
```

### Note

The memory layout of a quaternion is `xyzw`, i.e. the real part is stored last.

`core:math/linalg` contains many more procedures for working with quaternions, such as `quaternion_angle_axis`, `quaternion_slerp`, and `quaternion_mul_vector3`.

## SIMD vector types

The `#simd[N]T` type is a vector of `N` elements of type `T` which maps directly onto the SIMD (Single Instruction, Multiple Data) registers of the target where possible. `N` must be a power of two (up to 64), and `T` must be an integer, float, boolean, or `rawptr` type.

```
import "core:simd"

a := #simd[4]f32{1, 2, 3, 4}
b: #simd[4]f32 = 10 // a scalar is broadcast to every lane
c := a*b + a // arithmetic is applied lane-wise: <11, 22, 33, 44>

x := simd.extract(c, 2) // 33
total := simd.reduce_add_ordered(c) // 110
arr := simd.to_array(c) // [4]f32{11, 22, 33, 44}

mask := simd.lanes_lt(a, 3) // lane-wise comparison
d := simd.select(mask, a, b) // <1, 2, 10, 10>
e := cast(#simd[4]i32)a // lane-wise conversion
```

Unlike fixed arrays, individual lanes of a SIMD vector cannot be indexed directly or accessed with fields like `.x`. Use `simd.extract` and `simd.replace` for single lanes, and `swizzle` (or `simd.shuffle`) for rearranging lanes. Conditions (including those of ternary expressions) must be booleans; use `simd.select` for lane-wise selection.

The `core:simd` package (and the architecture-specific `core:simd/x86` and `core:simd/arm` packages) provides the full set of operations, along with common vector type aliases such as `simd.f32x4` and `simd.u8x16`. The underlying operations are the `simd_*` procedures in `base:intrinsics`.

## raw\_data procedure

`raw_data` is a built-in procedure which returns the underlying data of a built-in data type as a [Multi-Pointer](#).

```
raw_data([]$E) -> [^]E // slices
raw_data([dynamic]$E) -> [^]E // dynamic arrays
raw_data(^[$N]$E) -> [^]E // fixed array and enumerated arrays
raw_data(^#simd[$N]$E) -> [^]E // simd vectors
raw_data(^matrix[$I, $J]$E) -> [^]E // matrices
raw_data(string) -> [^]byte //
```

## using statement

`using` can be used to bring entities declared in a scope/namespace into the current scope. It can be applied to struct fields, and (as an opt-in feature) as a statement or to procedure parameters.

### using with struct fields

Let's take a very simple entity struct:

```
Vector3 :: struct{x, y, z: f32}
Entity :: struct {
 position: Vector3,
 orientation: quaternion128,
}
```

It can be used like this:

```
foo :: proc(entity: ^Entity) {
 fmt.println(entity.position.x, entity.position.y, entity.position.z)
}
```

`using` can be applied to struct fields, making all the fields of `position` appear as if they are on `Entity` itself:

```
Entity :: struct {
 using position: Vector3,
 orientation: quaternion128,
}
foo :: proc(entity: ^Entity) {
 fmt.println(entity.x, entity.y, entity.z)
}
```

### using as a statement

#### Note

`using` as a statement, or applied to a procedure parameter, is disallowed by default as it is considered bad practice outside of immediate refactoring. It can be enabled on a per-file basis with `#+feature using-stmt`. `using` on struct fields is always allowed.

```
#+feature using-stmt
package example

import "core:fmt"

bar :: proc() {
 // brings all the exported entities from the `fmt` package into this scope
 using fmt
 println("Hellope")
}
```

The entity members can be brought into the procedure scope by `using` it:

```
foo :: proc(entity: ^Entity) {
 using entity
 fmt.println(position.x, position.y, position.z)
}
```

The `using` can be applied to the parameter directly:

```
foo :: proc(using entity: ^Entity) {
 fmt.println(position.x, position.y, position.z)
}
```

It can also be applied to sub-fields:

```
foo :: proc(entity: ^Entity) {
 using entity.position
 fmt.println(x, y, z)
}
```

## Subtype polymorphism

It is possible to get subtype polymorphism, similar to inheritance-like functionality in C++, but without the requirement of vtables or unknown struct layout:

```
foo :: proc(entity: Entity) {
 fmt.println(entity.x, entity.y, entity.z)
}

Frog :: struct {
 ribbit_volume: f32,
 using entity: Entity,
}

frog: Frog
// Both work
frog.x = 123
foo(frog)
```

### Note

using can be applied to arbitrarily many things, which allows the ability to have multiple subtype polymorphism (but also its issues).

### Note

using'd fields can still be referred by name.

If the subtyping behaviour is wanted without bringing the field names into the parent struct, the #subtype field directive can be used instead of using:

```
Frog :: struct {
 ribbit_volume: f32,
 #subtype entity: Entity,
}

frog: Frog
frog.entity.x = 123 // the fields of `entity` must be accessed through `entity`
foo(frog) // but `Frog` can still be used where an `Entity` is expected
```

This is commonly used for COM-style interface hierarchies, e.g. in `vendor:directx`.

A procedure that takes a pointer to a struct can also be assigned to a procedure type which takes a pointer to one of that struct's subtypes, as long as that subtype field is at offset 0. This is useful for implementing vtables:

```
IFoo :: struct {
 foo: proc(self: ^IFoo) -> string,
}

Foo :: struct {
 using vtable: IFoo, // at offset 0
 name: string,
}

F00_VTABLE :: IFoo{
 // `self` is `^Foo` rather than `^IFoo`
 foo = proc(self: ^Foo) -> string {
 return self.name
 },
}

f := Foo{vtable = F00_VTABLE, name = "Hellope"}
fmt.println(f->foo()) // Hellope
```

## or\_else expression

or\_else is an infix binary operator that allows the user to define default values for certain expressions with optional-ok semantics.

```

m: map[string]int
i: int
ok: bool

if i, ok = m["hellope"]; !ok {
 i = 123
}
// The above can be mapped to `or_else`
i = m["hellope"] or_else 123

assert(i == 123)

```

`or_else` can be used with type assertions too, as they have optional-ok semantics.

```

v: union{int, f64}
i: int
i = v.(int) or_else 123
i = v.? or_else 123 // Type inference magic
assert(i == 123)

m: Maybe(int)
i = m.? or_else 456
assert(i == 456)

```

## or\_return operator

The concept of `or_return` will work by popping off the end value in a multiple valued expression and checking whether it was not `nil` or was `false`, and if so, sets the procedure's end return value to that value (if possible) and returns. If the procedure only has one return value, it will do a simple return. If the procedure had multiple return values, `or_return` will require that all results be named so that the end value could be assigned to by name and then an empty return could be called.

```

Error :: enum {
 None,
 Something_Bad,
 Something_Worse,
 The_Worst,
 Your_Mum,
}

caller_1 :: proc() -> Error {
 return .None
}

caller_2 :: proc() -> (int, Error) {
 return 123, .None
}

caller_3 :: proc() -> (int, int, Error) {
 return 123, 345, .None
}

foo_1 :: proc() -> Error {
 // This can be a common idiom in many code bases
 n0, err := caller_2()
 if err != nil {
 return err
 }

 // The above idiom can be transformed into the following
 n1 := caller_2() or_return

 // And if the expression is 1-valued, it can be used like this
 caller_1() or_return
 // which is functionally equivalent to
 if err1 := caller_1(); err1 != nil {
 return err1
 }

 // Multiple return values still work with `or_return` as it only
 // pops off the end value in the multi-valued expression
 n0, n1 = caller_3() or_return

 return .None
}

foo_2 :: proc() -> (n: int, err: Error) {
 // It is more common that your procedure returns multiple values
 // If `or_return` is used within a procedure that returns multiple
 // values (2+), then all the returned values must be named
 // so that a bare `return` statement can be used

 // This can be a common idiom in many code bases
 x: int
 x, err = caller_2()
 if err != nil {
 return
 }
}

```

```

}

// The above idiom can be transformed into the following
y := caller_2() or_return
_ = y

// And if the expression is 1-valued, it can be used like this
caller_1() or_return

// which is functionally equivalent to
if err1 := caller_1(); err1 != nil {
 err = err1
 return
}

// If using a non-bare `return` statement is required, setting the return values
// using the normal idiom is a better choice and clearer to read
if z, zerr := caller_2(); zerr != nil {
 return -345 * z, zerr
}

n = 123
return
}

caller_4 :: proc() -> (n: int, ok: bool) {
 return 3, true
}

foo_3 :: proc() -> (ok: bool) {
 // `or_return` also supports the ok semantics common in Odin code.
 // Note an error is indicated by `ok` being `false`
 x := caller_4() or_return

 if (x < 5) {
 ok = true
 }

 return
}

```

## or\_continue operator

`or_continue` is an operator that allows for simplified control flow in a loop. It functions much like a combination of `or_return` and `continue` in that it checks if the last value in a multiple-valued expression is `false` or `not nil`, and if so, behaves like a `continue` statement.

```

import "core:fmt"

Job :: struct {
 id: int,
 tasks_left: int,
 result: int,
 error_has_occurred: bool,
}

// Returns true if it completed its last task.
do_work_until_done :: proc(job: ^Job) -> bool {
 if job.tasks_left == 0 {
 return false
 }

 job.result *= 3
 job.tasks_left -= 1
 return job.tasks_left == 0
}

// Returns a result and true if everything worked out.
// Otherwise, returns zero and false.
tally :: proc(job: ^Job) -> (int, bool) {
 if job.error_has_occurred {
 return 0, false
 }

 return job.result, true
}

main :: proc() {
 jobs: [dynamic]Job
 append(&jobs, Job{ id = 1, })
 append(&jobs, Job{ id = 3, tasks_left = 2, result = 7 })
 append(&jobs, Job{ id = 5, tasks_left = 1, result = 5 })
 append(&jobs, Job{ id = 7, tasks_left = 1, result = 5, error_has_occurred = true })

 job_loop: for &job in jobs {
 do_work_until_done(&job) or_continue job_loop
 result := tally(&job) or_continue job_loop

 fmt.println("Job #i has completed its work. Result is: %i", job.id, result)
 }
}

```

```
}
}
```

The example above will print the following:

```
Job #5 has completed its work. Result is: 15
```

Only job #5 completes, because it has only 1 task left and its `error_has_occurred` is the default value, which is `false` in this case, as every variable in Odin is implicitly zeroed out unless made explicit.

Without `or_continue`, the above loop would look more like this:

```
// [...]
job_loop: for &job in jobs {
 if !do_work_until_done(&job) {
 continue job_loop
 }
 result, ok := tally(&job)
 if !ok {
 continue job_loop
 }

 fmt.Println("Job #i has completed its work. Result is: %i", job.id, result)
}
// [...]
```

The underlying operation is the same, but the code is more verbose where it doesn't need to be. As shown in the example, `or_continue` also works with labels, just like `continue`.

## or\_break operator

`or_break` follows the same mechanics as `or_continue`, except that it performs a `break` operation on its containing loop instead of a `continue`.

## Conditional compilation

A couple of ways are provided for doing this, and each of them have their uses.

### File suffixes

Often, you want to separate multiple implementations of a package based on the OS or the architecture. Your `.odin` files can have a magic suffix that will cause the compiler to either include or exclude them based on the target platform or architecture, or both.

For example, `foobar_windows.odin` would only be compiled on Windows, `foobar_linux.odin` only on Linux, and `foobar_windows_amd64.odin` only on Windows AMD64.

### when statements

Sometimes you only want to compile a block of code if a certain compile-time expression evaluates to `true`. This can be done using the [when statements](#):

```
when ODIN_OS == .Linux {
 // Do Linux stuff
}
```

The compiler provides a set of builtin constants which are available in all files in a compilation, and which can be used in a `when` condition. Here is a comprehensive list of them:

| Name             | Description                                                                                                                                                                                                                                    |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ODIN_ARCH        | An enum value indicating what the CPU architecture of the target is. ( <code>.amd64</code> , <code>.i386</code> , <code>.arm32</code> , <code>.arm64</code> , <code>.wasm32</code> , <code>.wasm64p32</code> , <code>.riscv64</code> )         |
| ODIN_ARCH_STRING | A string indicating what the CPU architecture of the target is. (" <code>amd64</code> ", " <code>i386</code> ", " <code>arm32</code> ", " <code>arm64</code> ", " <code>wasm32</code> ", " <code>wasm64p32</code> ", " <code>riscv64</code> ") |
| ODIN_BEDROCK     | <code>true</code> if the <code>-bedrock</code> command line switch is passed, which enables the restricted <a href="#">bedrock mode</a> .                                                                                                      |
| ODIN_BUILD_MODE  | An enum value indicating what type of compiled output the user desires. ( <code>.Executable</code> , <code>.Dynamic</code> , <code>.Static</code> , <code>.Object</code> , <code>.Assembly</code> , OR <code>.LLVM_IR</code> )                 |

| Name                            | Description                                                                                                                                                                                                                         |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ODIN_BUILD_PROJECT_NAME         | Name of the folder that contains the entry point.                                                                                                                                                                                   |
| ODIN_COMPILE_TIMESTAMP          | An i64 containing the time at which the executable was compiled, in nanoseconds. This is compatible with the <code>time.Time</code> type, i.e. <code>time.Time{_nsec=ODIN_COMPILE_TIMESTAMP}</code>                                 |
| ODIN_DEBUG                      | true if the <code>-debug</code> command line switch is passed, which enables debug info generation.                                                                                                                                 |
| ODIN_DEFAULT_CALLING_CONVENTION | An enum value of type <code>runtime.Calling_Convention</code> indicating the default <a href="#">calling convention</a> of procedures. ( <code>.Odin</code> )                                                                       |
| ODIN_DEFAULT_TO_NIL_ALLOCATOR   | true if the <code>-default-to-nil-allocator</code> command line switch is passed, which sets the initial allocator to an allocator that does nothing.                                                                               |
| ODIN_DEFAULT_TO_PANIC_ALLOCATOR | true if the <code>-default-to-panic-allocator</code> command line switch is passed, which sets the initial allocator to an allocator that panics if allocated from.                                                                 |
| ODIN_DISABLE_ASSERT             | true if the <code>-disable-assert</code> command line switch is passed, which removes all calls to <code>assert</code> from the compilation.                                                                                        |
| ODIN_ENDIAN                     | An enum value indicating the endianness of the target. ( <code>.Little</code> , <code>.Big</code> )                                                                                                                                 |
| ODIN_ENDIAN_STRING              | A string indicating the endianness of the target. (" <code>little</code> ", " <code>big</code> ")                                                                                                                                   |
| ODIN_ERROR_POS_STYLE            | An enum value set using the <code>-error-pos-style</code> switch, indicating the source location style used for compile errors and warnings. ( <code>.Default</code> ( <code>Odin</code> ), <code>.Unix</code> )                    |
| ODIN_FOREIGN_ERROR_PROCEDURES   | true if the <code>-foreign-error-procedures</code> command line switch is passed, which inhibits generation of runtime error procedures; that they can be in a separate compilation unit.                                           |
| ODIN_MICROARCH_STRING           | A string describing the microarchitecture used for code generation. Can be set using the <code>-microarch</code> command line switch. E.g. " <code>sandybridge</code> ".                                                            |
| ODIN_MINIMUM_OS_VERSION         | An integer value representing the minimum OS version set using <code>-minimum-os-version</code> . Calculated as <code>major * 10_000 + minor * 100 + revision</code> , and defaults to 0 if not specified.                          |
| ODIN_NO_BOUNDS_CHECK            | true if the <code>-no-bounds-check</code> command line switch is passed, which disables bounds checking at runtime.                                                                                                                 |
| ODIN_NO_CRT                     | true if the <code>-no-crt</code> command line switch is passed, which inhibits linking with the C Runtime Library, a.k.a. <code>LibC</code> .                                                                                       |
| ODIN_NO_ENTRY_POINT             | true if the <code>-no-entry-point</code> command line switch is passed, which makes the declaration of a <code>main</code> procedure optional.                                                                                      |
| ODIN_NO_RTTI                    | true if the <code>-no-rtti</code> command line switch is passed, which inhibits generation of full Runtime Type Information.                                                                                                        |
| ODIN_NO_TYPE_ASSERT             | true if the <code>-no-type-assert</code> command line switch is passed, which disables type assertion checking program wide.                                                                                                        |
| ODIN_OPTIMIZATION_MODE          | An enum value indicating the optimization level selected using the <code>-o</code> command line switch. ( <code>.None</code> , <code>.Minimal</code> , <code>.Size</code> , <code>.Speed</code> , <code>.Aggressive</code> )        |
| ODIN_OS                         | An enum value indicating what the target operating system is.                                                                                                                                                                       |
| ODIN_OS_STRING                  | A string indicating what the target operating system is.                                                                                                                                                                            |
| ODIN_PLATFORM_SUBTARGET         | An enum value indicating the platform subtarget, chosen using the <code>-subtarget</code> switch. ( <code>.Default</code> , <code>.iPhone</code> , <code>.iPhoneSimulator</code> , <code>.Android</code> , <code>.Playdate</code> ) |
| ODIN_ROOT                       | Path to the folder containing the Odin compiler executable.                                                                                                                                                                         |
| ODIN_SANITIZER_FLAGS            | A <code>bit_set</code> indicating the sanitizer flags set using the <code>-sanitize</code> command line switch. ( <code>.Address</code> , <code>.Memory</code> , and <code>.Thread</code> ).                                        |
| ODIN_TEST                       | true if the code is being compiled via an invocation of <code>odin test</code> .                                                                                                                                                    |

| Name                          | Description                                                                                                                                                                                                                                         |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ODIN_USE_SEPARATE_MODULES     | true by default, false if the <code>-use-single-module</code> command line switch is passed to force a unity build. By default each package is compiled into an object file, then linked together.                                                  |
| ODIN_VALGRIND_SUPPORT         | true if Valgrind integration is supported on the target.                                                                                                                                                                                            |
| ODIN_VENDOR                   | String which identifies the compiler being used. The official compiler sets this to "odin".                                                                                                                                                         |
| ODIN_VERSION                  | A string that represents the Odin compiler version being used. (e.g: dev-2023-04)                                                                                                                                                                   |
| ODIN_VERSION_HASH             | A string containing the Git hash part of the Odin version. Empty if <code>.git</code> could not be detected at the time the compiler was built.                                                                                                     |
| ODIN_WINDOWS_SUBSYSTEM        | An enum value indicating the desired Windows PE subsystem, chosen using the <code>-subsystem</code> switch. (e.g. <code>.Console</code> (default), <code>.Windows</code> , <code>.EFI_Application</code> , or <code>.Unknown</code> on non-Windows) |
| ODIN_WINDOWS_SUBSYSTEM_STRING | A string indicating the desired Windows PE subsystem, chosen using the <code>-subsystem</code> switch. (e.g. "CONSOLE" (default), "WINDOWS", or "" on non-Windows platforms).                                                                       |
| __ODIN_LLVM_F16_SUPPORTED     | true if LLVM supports the f16 type.                                                                                                                                                                                                                 |

### Note

The `.Playdate` subtarget targets the [Playdate](#) handheld, and is used together with the `freestanding_arm32` target: `-target:freestanding_arm32 -subtarget:playdate`.

See the [tracking allocator](#) for an example of something that uses the `ODIN_DEBUG` constant in a `when` statement.

## Command-line defines

Sometimes you want to do something conditionally based on some compile-time parameters of some sort, but globally, across the entire project. This is how you define those.

You may define a constant using the `-define` command line switch. e.g: `-define:F00=true`. You can then fetch its value as a constant in your code like this:

```
F00 :: #config(F00, false) // defines `F00` as a constant with the default value of false
BAR :: #config(BAR_DEBUG, true) // name can be different compared to the constant

when F00 {
 // only evaluated when `F00` is true
} else {
 // only evaluate when `F00` is false
}
```

The value for a command line define may be an integer, boolean, or string. Currently, no other types are supported.

You can read up further on Built-in procedures [here](#).

## Build tags

This feature allows you to cover more edge-case situations where you want some code to be compiled on several platforms.

However, overly-liberal use of this feature can make it hard to reason about what code is included or not, based on the target platform or architecture. [File Suffixes](#) are typically a nicer approach if they cover what you need.

For the sake of demonstration, let's take POSIX: You could use `foobar_unix.odin`, which has no special meaning to the compiler at all, and use a tag in the file itself.

Here's an example of a file that will only be included on Linux or Darwin:

```
##build linux, darwin
package foobar
```

The opposite, excluding the file on both Linux and Darwin, is achieved like this:

```
##build !linux
##build !darwin
package foobar
```

bedrock can also be used as a build tag, to include or exclude a file when compiling in [bedrock mode](#):

```
##build !bedrock
package foobar
```

## Advanced build tags

### ##vet

Can be used to enable or disable vetting options on a per-file basis.

Possible Options:

- unused
- unused-variables
- unused-imports
- shadowing
- using-stmt
- using-param
- style
- semicolon
- deprecated
- cast
- tabs
- unused-procedures
- explicit-allocators
- when-shadowing - declarations within a global `when` which shadow a built-in or package-level name

### ##test

The file is completely ignored from parsing and type checking *EXCEPT* during `odin test`.

### ##ignore

The file is completely ignored from parsing and type checking.

### ##build-project-name

Includes the file only if the name of the project being built (`ODIN_BUILD_PROJECT_NAME`, the name of the directory containing the entry point) matches. Like `##build`, comma separated groups are alternatives, and names can be negated with `!`:

```
##build-project-name my_game, my_game_tests
package foobar
```

```
##build-project-name !my_editor
package foobar
```

### ##private

Using `##private` before the package declaration will automatically add `@(private)` to everything in that file:

```
##private
package foo
```

And `##private` file will be equivalent to automatically adding `@(private="file")` to each declaration. This means that to remove the private-to-file association, you must apply a private-to-package attribute `@(private)` to the declaration.

### ##feature

Enables a specific feature or changes the behaviour of a specific aspect of the language

- `##feature dynamic-literals`
  - Enables dynamic-literals for types `map` and `[dynamic]T`
- `##feature integer-division-by-zero:<option>`

- Makes all integer division within this file operation a specific way
- `#+feature integer-division-by-zero:trap --` Trap on division/modulo/remainder by zero
- `#+feature integer-division-by-zero:zero --`  $x/0 == 0$  and  $x\%0 == x$  and  $x\%0 == x$
- `#+feature integer-division-by-zero:self --`  $x/0 == x$  and  $x\%0 == 0$  and  $x\%0 == 0$
- `#+feature integer-division-by-zero:all-bits --`  $x/0 == \sim T(0)$  and  $x\%0 == x$  and  $x\%0 == x$
- `#+feature global-context`
  - Enable the use of globals that require the `context` to be defined in the global scope
  - The `context` by default does not exist in the global scope
- `#+feature using-stmt`
  - Enables the use of `using` as a statement and on procedure parameters within this file (see [using as a statement](#))
- `#+feature force-type-assert`
  - Keeps type assertions enabled within this file, even if `-no-type-assert` is passed

#### `#+no-instrumentation`

Disables instrumentation within the entire file

#### `#+lazy`

**For internal use.** Declarations in this file are only type checked if they are used.

## Bedrock mode

The `-bedrock` command line switch enables a restricted subset of Odin, which is useful for very low level and freestanding code where the full runtime is not wanted. In bedrock mode the following are disabled:

- map types
- 128-bit integer types
- Runtime type information (as if `-no-rtti` was passed)
- Global variables which are not initialized with constants (as if `-disable-non-constant-globals` was passed)
- `@(init)` and `@(fini)` procedures (as if `-disable-init-fini` was passed)
- Anything Objective-C related
- The default paths to the `core` and `vendor` library collections

The default allocator is also the nil allocator (as if `-default-to-nil-allocator` was passed), and `main` may only use the "odin" or "contextless" calling conventions.

Code can check for bedrock mode with the `ODIN_BEDROCK` constant, or with the [bedrock build tag](#).

## Implicit context system

In each scope, there is an implicit value named `context`. This `context` variable is local to each scope and is implicitly passed by pointer to any procedure call in that scope (if the procedure has the Odin calling convention).

The main purpose of the implicit `context` system is for the ability to intercept third-party code and libraries and modify their functionality. One such case is modifying how a library allocates something or logs something. In C, this was usually achieved with the library defining macros which could be overridden so that the user could define what they wanted. However, not many libraries supported this in many languages by default which meant intercepting third-party code to see what it does and to change how it does it was not possible.

```
main :: proc() {
 c := context // copy the current scope's context

 context.user_index = 456
 {
 context.allocator = my_custom_allocator()
 context.user_index = 123
 supertramp() // the `context` for this scope is implicitly passed to `supertramp`
 }

 // `context` value is local to the scope it is in
 assert(context.user_index == 456)
}

supertramp :: proc() {
 c := context // this `context` is the same as the parent procedure that it was called from
 // From this example, context.user_index == 123
}
```

```
// A context allocator is assigned to the return value of `my_custom_allocator()`

// The memory management procedure uses the `context.allocator` by default unless explicitly specified otherwise
ptr := new(int)
free(ptr)
}
```

By default, the `context` value has default values for its parameters which is decided in the package runtime. These defaults are compiler specific.

To see what the implicit `context` value contains, please see the definition of the `Context` struct in [package runtime](#).

Limitation: Because of the way the copy-on-write `context` system works and is tied to scopes, you can't update a context value (like e.g. `context.user_index`) from within a loop. [If you need to update something](#) on the context within a loop to be available to callees, try setting `context.user_ptr` to some user data outside the loop, and modify that user data from within the loop.

## Allocators

Odin is a manual memory management based language. This means that Odin programmers must manage their own memory, allocations, and tracking. To aid with memory management, Odin has huge support for custom allocators, especially through the implicit `context` system.

The built-in types of dynamic arrays and `map` both contain a custom allocator. This allocator can be either manually set or the allocator from the current `context` will be assigned to the data type.

All allocations in Odin are preferably done through allocators. The core library of Odin takes advantage of allocators through the implicit `context` system. The following call:

```
ptr := new(int)
```

is equivalent to this:

```
ptr := new(int, context.allocator)
```

The allocator from the `context` is implicitly assigned as a default parameter to the built-in procedure `new`.

The implicit `context` stores two different forms of allocators: `context.allocator` and `context.temp_allocator`. Both can be reassigned to any kind of allocator. However, these allocators are to be treated slightly differently.

- `context.allocator` is for "general" allocations, for the subsystem it is used within.
- `context.temp_allocator` is for temporary and short lived allocations, which are to be freed once per cycle/frame/etc.

By default, the `context.allocator` is an OS heap allocator and the `context.temp_allocator` is assigned to a scratch allocator (a growing arena based allocator). `free_all(context.temp_allocator)` must be called to clear the contents of the temporary allocator's internal arena.

The following procedures are built-in (and also available in `package mem` with enforced allocator errors) and are encouraged for managing memory:

- `new` - allocates a value of the type given. The result value is a pointer to the type given.

```
ptr := new(int)
ptr^ = 123
x: int = ptr^
```

- `new_clone` - allocates a clone of the value passed to it. The resulting value of the type will be a pointer to the type of the value passed.

```
x: int = 123
ptr: ^int
ptr = new_clone(x)
assert(ptr^ == 123)
```

- `make` - allocates memory for a backing data structure of either a [slice](#), [dynamic array](#), or [map](#).

```
slice := make([int], 65)

dynamic_array_zero_length := make([dynamic]int)
dynamic_array_with_length := make([dynamic]int, 32)
dynamic_array_with_length_and_capacity := make([dynamic]int, 16, 64)
```

```
made_map := make(map[string]int)
made_map_with_reservation := make(map[string]int, 64)
```

- `new_aligned` and `make_aligned` - like `new` and `make` (for slices) but with an explicitly specified alignment.

```
ptr, err := new_aligned(int, 64) // aligned to 64 bytes
data := make_aligned([]f32, 1024, 64) // 1024 elements, aligned to 64 bytes
```

- `free` - frees the memory at the pointer given. **Note:** only free memory with the allocator it was allocated with.

```
ptr := new(int)
free(ptr)
```

- `free_all` - frees all the memory of the context's allocator (or given allocator). **Note:** not all allocators support this procedure.

```
free_all()
free_all(context.temp_allocator)
free_all(my_allocator)
```

- `delete` - deletes the backing memory of a value allocated with `make` or a string that was allocated through an allocator.

```
delete(my_slice)
delete(my_dynamic_array)
delete(my_map)
delete(my_string)
delete(my_cstring)
```

To see more uses of allocators and allocation-related procedures, please see [package mem](#) or its sub-packages in the core library.

For more information regarding memory allocation strategies in general, please see [Ginger Bill's Memory Allocation Strategy](#) series.

## Tracking allocator

In the core collection you'll find a tracking allocator that warns you if your program is leaking memory or if it does bad frees. Here's how to set it up:

```
package main

import "core:fmt"
import "core:mem"

main :: proc() {
 when ODIN_DEBUG {
 track: mem.Tracking_Allocator
 mem.tracking_allocator_init(&track, context.allocator)
 context.allocator = mem.tracking_allocator(&track)

 defer {
 if len(track.allocation_map) > 0 {
 fmt.eprintf("=== %v allocations not freed: ===\n", len(track.allocation_map))
 for _, entry in track.allocation_map {
 fmt.eprintf("- %v bytes @ %v\n", entry.size, entry.location)
 }
 }
 mem.tracking_allocator_destroy(&track)
 }
 }
 do_stuff()
}
```

This uses a [when statement](#) to only enable the tracking allocator when the `-debug` compilation flag is set. Since it sets the tracking allocator on the context in the beginning of `main`, the rest of the program will use this tracking allocator.

Note that `when` blocks do not have a real scope, the curly braces `{}` are just there to group code. Any changes to the `context` within a `when` block are valid after the `when` block ends.

By default, the tracking allocator will panic if a bad free occurs. It will print a message that informs you where that bad free happened. You can override that behavior by overriding `track.bad_free_callback`:

```
// This will add the bad frees to `track.bad_free_array`,
// you must manually check the contents of that array.
track.bad_free_callback = mem.tracking_allocator_bad_free_callback_add_to_array
```

## Explicit context definition

Procedures which do not use the "odin" calling convention must explicitly assign the `context` if something within its body requires it.

```
explicit_context_definition :: proc "c" () {
 // Try commenting the following statement out below
 context = runtime.default_context()

 fmt.println("\n#explicit context definition")
 dummy_procedure()
}

dummy_procedure :: proc() {
 fmt.println("dummy_procedure")
}
```

Here is another example of setting an error callback for `vendor:glfw`:

```
error_callback :: proc "c" (code: i32, desc: cstring) {
 context = runtime.default_context() // set the current context
 fmt.println(desc, code) // fmt.* calls use the odin calling convention
}
glfw.SetErrorCallback(error_callback)
```

## Logging system

As part of the implicit `context` system, there is a built-in logging system.

To see more uses of loggers, please see [package log](#) in the core library.

## Foreign system

It is sometimes necessary to interface with foreign code, such as a C library. In Odin, this is achieved through the `foreign` system. You can "import" a library into the code using the same semantics as a normal import declaration:

```
foreign import kernel32 "system:kernel32.lib"
```

This `foreign import` declaration will create a "foreign import name" which can then be used to associate entities within a foreign block.

The path of a `foreign import` may be relative to the current file, an absolute path, a system library using the `system: prefix`, or within a library collection, including custom collections defined with the `-collection:<name>=<path>` command line switch. Multiple paths can be given within braces:

```
foreign import foo "libfoo.a" // relative to this file
foreign import bar "system:bar" // a system library, e.g. `libbar.so`, `bar.lib`
foreign import baz "shared:libbaz.a" // with `-collection:shared=path/to/shared`
foreign import qux {"qux.lib", "qux_extra.lib"}
```

```
foreign import kernel32 "system:kernel32.lib"
foreign kernel32 {
 ExitProcess :: proc "stdcall" (exit_code: u32) ---
}
```

The Odin compiler can also automatically build and link imported assembly files. Recognized file extensions for assembly files are: `asm`, `s`, and `S`. Depending on the target, a different assembler is used to compile the assembly:

- Windows: [nasm](#) (which is shipped with the Odin compiler)
- macOS: `as` (which ships with macOS)
- Other platforms on arm64 (AArch64) and riscv64: [clang](#)
- Other platforms on amd64 and i386: [nasm](#) (which needs to be installed separately)

Extra flags can be passed to the assembler with the `-extra-assembler-flags:<string>` command line switch.

For examples of how Odin uses this feature, see `base/runtime/entry_*.asm`.

```
foreign import lowlevel "lowlevel.asm"
```

```
foreign lowlevel {
 __get_flags :: proc "c" () -> u64 ---
}
```

```
bits 64

global __get_flags

section .text
__get_flags:
 pushfq
 pop rax
 ret
```

If a library exports global variables, you can import those into Odin as well.

```
foreign lib {
 x: i32
}
```

Foreign procedure declarations have the **cdecl/c** calling convention by default unless specified otherwise. Due to foreign procedures not having a body declared within this code, you need to append the `---` symbol to the end to distinguish it as a procedure literal without a body and not a procedure type.

The attributes system can be used to change specific properties of entities declared within a block:

```
@(default_calling_convention = "std")
foreign kernel32 {
 @(link_name="GetLastError") get_last_error :: proc() -> i32 ---
}
```

Available attributes for foreign blocks:

- `default_calling_convention=<string>` - The default calling convention for procedures declared within this foreign block.
- `link_prefix=<string>` - This prefix is prepended to the linkage names of the entities except where the link name has been explicitly overridden.
- `link_suffix=<string>` - This suffix is appended to the linkage names of the entities except where the link name has been explicitly overridden.
- `private=<string>` - The default private level for all entities. Defaults to "package" if not set but may be set to "file".
- `require_results` - All procedures declared within this foreign block must have their return values used.

Available attributes for foreign import declarations:

- `extra_linker_flags=<string>` - Specifies additional linker flags for this foreign import.
- `export` - Makes the foreign import name visible to every file in the package, rather than just the current file.
- `require` - Forces the library to be linked, even if none of its entities are used.
- `priority_index=<integer>` - Controls the order in which libraries are passed to the linker; libraries with a lower priority index are passed first.
- `ignore_duplicates` - Skips passing the library to the linker again if it has already been passed.

```
@(export, extra_linker_flags="/NODEFAULTLIB:libcmt")
foreign import lib "libfoo.lib"
```

## Implementing C variadic procedures

`#c_vararg` is not only for calling foreign variadic procedures. A procedure with the "c" calling convention can also implement a C-style variadic procedure in Odin, reading its arguments with the `c_va_*` procedures from `base:intrinsics` (which are also available as `va_start`, `va_arg`, etc. in `core:c/libc`):

```
import "base:intrinsics"
import "core:c"

c_sum :: proc "c" (count: c.int, #c_vararg args: ..any) -> (total: c.int) {
 list: c.va_list
 intrinsics.c_va_start(&list, args)
 defer intrinsics.c_va_end(&list)
 for _ in 0..

```

**Note**

As in C, nothing is known about the number or the types of the variadic arguments; they must be determined by other means (such as `count` above, or a format string), and read in the order in which they were passed.

## Inline assembly

Odin has experimental support for inline assembly through `asm` templates, on the amd64, arm64, and riscv64 targets. An `asm` template is a callable entity which is instantiated in place at each call, like a forced-inline procedure, and it uses a single syntax across all instruction set architectures:

```
add_one :: asm(x: u64) -> (r: u64) [x -> r] { inc r }

mfence :: asm() [#volatile, #clobber memory] { mfence }

y := add_one(41) // 42
```

Please see the dedicated inline `asm` templates documentation for more information.

## Using a vendor library

As described in the [Foreign System](#) we often want to use existing c libraries. Odin has a wide collection of maintained [bindings and ports](#).

**Note**

Case notation should remain the same as the original authors intended, to make porting code easier.

Let's run through how we could use the `vendor:glfw` library. The code will be based on their [Quick Guide](#) but we will simplify it to only show using `glfw`.

```
package main

import "base:runtime"
import "core:fmt"
import "vendor:glfw"

error_callback :: proc "c" (code: i32, desc: cstring) {
 context = runtime.default_context()
 fmt.println(desc, code)
}

key_callback :: proc "c" (window: glfw.WindowHandle, key, scancode, action, mods: i32) {
 if key == glfw.KEY_ESCAPE && action == glfw.PRESS {
 glfw.SetWindowShouldClose(window, glfw.TRUE)
 }
}

main :: proc() {
 glfw.SetErrorCallback(error_callback)

 if !glfw.Init() {
 panic("EXIT_FAILURE")
 }
 defer glfw.Terminate()

 glfw.WindowHint(glfw.CONTEXT_VERSION_MAJOR, 2)
 glfw.WindowHint(glfw.CONTEXT_VERSION_MINOR, 0)

 window := glfw.CreateWindow(640, 480, "Simple example", nil, nil)
 if window == nil {
 panic("EXIT_FAILURE")
 }
 defer glfw.DestroyWindow(window)

 glfw.SetKeyCallback(window, key_callback)

 glfw.MakeContextCurrent(window)
 // ...
 glfw.SwapInterval(1)
 // ...

 for !glfw.WindowShouldClose(window) {
 // ...

 glfw.SwapBuffers(window)
 glfw.PollEvents()
 }
}
```

As we can see there is little difference to how someone would use `vendor:glfw` in this case. It's not always perfect but often good enough to port existing code quickly.

## Parametric polymorphism

Parametric polymorphism, commonly referred to as "generics", allow the user to create a procedure or data that can be written *generically* so it can handle values in the same manner.

### Note

Within the Odin code base and documentation, the nickname "parapoly" is usually used.

### Explicit parametric polymorphism

Explicit parametric polymorphism means that the types of the parameters of a `proc` or of the data fields of a `struct` (when intended to potentially be used with multiple possible types) must be explicitly provided. This is similar to how C++ allows the use of `templates` to fill out the body of a procedure or data structure with the types that are given at compile-time as input to the `template` parameters, but in Odin explicit parametric polymorphism is safer and cleaner to work with.

### Procedures using explicit parametric polymorphism (parapoly)

As a reminder, all parameters passed into a function are immutable in the sense that they can't have their value changed using `=` directly. A useful idiom is `var := var`, which expresses a variable shadowing itself. When used at the top of a procedure the compiler understands the use case of enabling local modification of the otherwise immutable parameter variable, and won't complain about the shadowing when you compile with `-vet`.

```
sin_tau :: proc(angle_in_cycles: f64) -> f64 {
 angle_in_cycles := angle_in_cycles // Allows `angle_in_cycles` to have its value changed

 TAU := 2 * math.PI
 angle_in_cycles *= TAU
 return math.sin(angle_in_cycles)
}
assert(math.abs(sin_tau(0.25) - 1) <= 0.001) // sin_tau(0.25) is approximately 1
assert(math.abs(sin_tau(0.75) - -1) <= 0.001) // sin_tau(0.75) is approximately -1
```

However, to specify that a parameter must be a **compile-time** constant, which is not the same thing as an immutable parameter, and may sometimes be necessary (e.g. for parapoly) or desirable (e.g. to enforce compile-time computation), the parameter's name must be prefixed with a dollar sign `$`. The following example takes two compile-time constant parameters and then uses them to initialize an array of known length:

```
make_f32_array :: #force_inline proc($N: int, $val: f32) -> (res: [N]f32) {
 for _, i in res {
 res[i] = val*val
 }
 return
}

array := make_f32_array(3, 2)
```

Types can also be explicitly passed by specifying that the `typeid` parameter is constant:

```
my_new :: proc($T: typeid) -> ^T {
 return (^T)(alloc(size_of(T), align_of(T)))
}

ptr := my_new(int)
```

### Data types using explicit parametric polymorphism (parapoly)

Structures and unions may have polymorphic parameters and the syntax for doing so is similar to procedure call syntax. **Parapoly struct:**

```
Table_Slot :: struct($Key, $Value: typeid) {
 occupied: bool,
 hash: u32,
 key: Key,
 value: Value,
}
slot: Table_Slot(string, int)
```

**Parapoly union:**

```
Error :: enum {Foo0, Foo1, Foo2}
Param_Union :: union($T: typeid) #no_nil {T, Error}
r: Param_Union(int)
r = 123
r = Error.Foo0
```

The \$ prefix is optional for record data types as all parameters must be "constant".

## Implicit parametric polymorphism

Implicit implies that the type of a parameter is inferred from its input. In this case, the dollar sign \$ can be placed on the type.

### Note

Within the Odin code base and documentation, the name "polymorphic name" is usually used.

## Procedures using implicit parametric polymorphism (parapoly)

```
foo :: proc($N: $I, $T: typeid) -> (res: [N]T) {
 // `N` is the constant value passed
 // `I` is the type of `N`
 // `T` is the type passed
 fmt.printf("Generating an array of type %v from the value %v of type %v\n",
 typeid_of(type_of(res)), N, typeid_of(I))
 for i in 0..<N {
 res[i] = i*i
 }
 return
}

T :: int
array := foo(4, T)
for v, i in array {
 assert(v == T(i*i))
}
```

## Specialization

In some cases, you may want to specify that a type must be a specialization of a certain type.

```
// Only allow types that are specializations of a (polymorphic) slice
make_slice :: proc($T: typeid/[]$E, len: int) -> T {
 return make(T, len)
}
```

```
Table_Slot :: struct($Key, $Value: typeid) {
 occupied: bool,
 hash: u32,
 key: Key,
 value: Value,
}
Table :: struct($Key, $Value: typeid) {
 count: int,
 allocator: mem.Allocator,
 slots: []Table_Slot(Key, Value),
}

// Only allow types that are specializations of `Table`
allocate :: proc(table: ^$T/Table, capacity: int) {
 ...
}

// find :: proc(table: ^$T/Table, key: T.Key) -> (T.Value, bool) {
find :: proc(table: ^Table($Key, $Value), key: Key) -> (Value, bool) {
 ...
}
```

## where clauses

A bound on polymorphic parameters to a procedure or record can be expressed using a `where` clause immediately before opening `{`, rather than at the type's or constant's first mention. Additionally, `where` clauses can apply bounds to arbitrary types, rather than just polymorphic type parameters.

Some cases that a `where` clause may be useful:

- Sanity checks for parameters:

```
simple_sanity_check :: proc(x: [2]int)
```

```

where len(x) > 1,
 type_of(x) == [2]int {
 fmt.println(x)
 }

```

- Parameter polymorphism checks for procedures:

```

cross_2d :: proc(a, b: $T/[2]$E) -> E
 where intrinsics.type_is_numeric(E) {
 return a.x*b.y - a.y*b.x
 }
cross_3d :: proc(a, b: $T/[3]$E) -> T
 where intrinsics.type_is_numeric(E) {
 x := a.y*b.z - a.z*b.y
 y := a.z*b.x - a.x*b.z
 z := a.x*b.y - a.y*b.x
 return T{x, y, z}
 }

a := [2]int{1, 2}
b := [2]int{5, -3}
fmt.println(cross_2d(a, b))

x := [3]f32{1, 4, 9}
y := [3]f32{-5, 0, 3}
fmt.println(cross_3d(x, y))

// Failure case
// i := [2]bool{true, false}
// j := [2]bool{false, true}
// fmt.println(cross_2d(i, j))

```

- Solving disambiguations with polymorphic procedures in a procedure grouping:

```

foo :: proc(x: [$N]int) -> bool
 where N > 2 {
 fmt.println(#procedure, "was called with the parameter", x)
 return true
 }

bar :: proc(x: [$N]int) -> bool
 where 0 < N,
 N <= 2 {
 fmt.println(#procedure, "was called with the parameter", x)
 return false
 }

baz :: proc{foo, bar}

x := [3]int{1, 2, 3}
y := [2]int{4, 9}
ok_x := baz(x)
ok_y := baz(y)
assert(ok_x == true)
assert(ok_y == false)

```

- Restrictions on parametric polymorphic parameters for record types:

```

Foo :: struct($T: typeid, $N: int)
 where intrinsics.type_is_integer(T),
 N > 2 {
 x: [N]T,
 y: [N-2]T,
 }

T :: i32
N :: 5
f: Foo(T, N)
#assert(size_of(f) == (N+N-2)*size_of(T))

```

## -> operator (selector call expressions)

The -> operator is called the selector call expression operator and is extremely useful for call procedures stored in vttables. [Component Objective Model \(COM\)](#) APIs is a great example of where this kind of thing is extremely useful (such as the [Direct3D11 package](#)).

```

x->y(123)
// is equivalent to
x.y(x, 123)

```

As the -> operator is effectively syntactic sugar, all of the same semantics still apply, meaning subtyping through using will still work as expected to allow for the emulation of type hierarchies.

## Attributes

Attributes modify the compilation details or behaviour of declarations. They are written before a declaration, as `@(name)` or `@(name=value)`, and one `@(...)` may hold several:

```
@(require_results)
make_vector :: proc(x, y: f32) -> [2]f32 {
 return {x, y}
}

@(private="file", rodata)
powers_of_two := [4]int{1, 2, 4, 8}

@(init)
startup :: proc "contextless" () {
 // runs before `main`
}
```

Commonly used ones include:

- `@(require_results)` -- the results of a call to the procedure must be used
- `@(private)` -- keeps a declaration within its package, or with "file", within its file
- `@(static)` and `@(thread_local)` -- a local variable which keeps its value between calls
- `@(rodata)` -- a variable in read-only memory
- `@(init)` and `@(fini)` -- procedures run before and after `main`
- `@(deferred_out=<procedure>)`, and the other `deferred_*` attributes -- a procedure called at the end of the caller's scope
- `@(link_name=<string>)` -- the name of a foreign or exported symbol
- `@(test)` -- a procedure run by `odin test`

See Attributes for all of them, and the declarations each one applies to.

## Directives

Directives are a way of extending the core behaviour of the Odin programming language. They have the form `#directive_name`, and appear on types, statements, procedure parameters, and expressions:

```
Header :: struct #packed {
 tag: u8,
 size: u32,
}
#assert(size_of(Header) == 5)

DEBUG :: #config(DEBUG, false)

logo := #load("logo.png") // the file's contents, baked into the program

Direction :: enum {North, East, South, West}

turn_right :: proc(d: Direction) -> Direction {
 #partial switch d {
 case .North: return .East
 case .South: return .West
 }
 return d
}

log_location :: proc(loc := #caller_location) {
 fmt.println(loc)
}
```

Commonly used ones include:

- `#packed`, `#raw_union`, and `#align` -- the memory layout of a struct
- `#partial` -- a switch which need not cover every case
- `#no_bounds_check` -- turns bounds checking off for a statement, block, or procedure
- `#force_inline` -- inlines a procedure or a call, always
- `#caller_location` -- a parameter which defaults to where the procedure was called from
- `#optional_ok` -- lets callers leave out a procedure's final `bool` result
- `#assert` -- an assertion checked at compile time
- `#config` -- a constant which can be set with `-define:NAME=VALUE`
- `#load` -- a file's contents, at compile time

See Directives for all of them.

## Useful idioms

The following are useful idioms which are emergent from the semantics of the language.

### Basic idioms

#### Ternary operator

The following two snippets are identical:

```
bar := condition ? 1 : 42
```

```
bar := 1 if condition else 42
```

You can also use ternary expressions with constants at compile-time:

```
DEBUG_LOG_SIZE :: 1024 when ODIN_DEBUG else 0
```

### If-statements with initialization

```
if str, ok := value.(string); ok {
 ...
} else {
 ...
}
```

### Iterating through slices of structs by value or by reference

```
Foo :: struct {
 f: f32,
 i: i32,
}

foos := make([]Foo, num)

// By-value basic range-based loop, with implicit indexing
for v, j in foos {
 fmt.Println(j, v, v.f, v.i)
}

// Alternative range-based loop, with explicit indexing
for _, j in foos {
 foo := foos[j] // copy
 fmt.Println(j, foo, foo.f, foo.i)
}

// By-reference range-based explicit indexing loop
for _, j in foos {
 foo := &foos[j] // "reference", changes to `foo.f` or `foo.i` are visible outside this scope
 fmt.Println(j, foo, foo.f, foo.i)
}

// By-reference range-based loop
for &v, j in foos {
 v.f += 1 // `v` is a reference to the element; changes are visible outside this scope
 fmt.Println(j, v, v.f, v.i)
}
```

#### defer if

The `defer if` idiom is equivalent to an `if` statement inside of a `defer`. It is merely a shorthand that comes about from the natural evaluation of `if` as a statement in its own right; it does not optionally defer a statement on the basis of a boolean condition at the time of evaluation, but it evaluates the condition once the deferred block is acted upon.

```
cond := false
defer {
 if cond { fmt.Println("c") } // This will print last.
}
defer if cond {
 fmt.Println("b") // This will print after "a".
}
defer {
 // This is first evaluated, allowing the prior `defer`s to act, as evaluation
 // happens in reverse declaration order.
 cond = true
 fmt.Println("a") // This will print first.
}
```

### Maybe(T)

Maybe(T) is a union which *either* holds a value of type T or is nil. In other languages this is often seen as Option(T), Result(T), etc.

Odin has multiple return values; Maybe(T) is used less frequently or rarely in the core library. Instead of doing -> Maybe(int) you could transform it to -> (int, bool).

```
halve :: proc(n: int) -> Maybe(int) {
 if n % 2 != 0 do return nil
 return n / 2
}

half, ok := halve(2).?
if ok do fmt.println(half) // 1
half, ok = halve(3).?
if !ok do fmt.println("3/2 isn't an int")

n := halve(4).? or_else 0
fmt.println(n) // 2
```

### Advanced idioms

Subtype polymorphism with run-time type-safe down-casting:

```
Entity :: struct {
 id: u64,
 name: string,
 variant: union{^Frog},
}

Frog :: struct {
 using entity: Entity,
 volume: f32,
 jump_height: i32,
}

new_entity :: proc($T: typeid) -> ^T {
 e := new(T)
 e.variant = e
 return e
}

entity: ^Entity = new_entity(Frog)
switch e in entity.variant {
case ^Frog:
 fmt.println("Ribbit:", e.volume)
}
```

### Implicit type conversions

Odin is a strongly and distinctly typed language by default. It has very few implicit type conversions compared to many other languages.

- ^T -> rawptr
- [^]T -> rawptr
- [^]T <-> ^T
- All types to any (must be specialized/non-polymorphic)
- Any of its variants to the union
- fN -> complex2N (e.g. f32 -> complex64)
- fN -> quaternion4N (e.g. f32 -> quaternion128)
- complex2N -> quaternion4N (e.g. complex64 -> quaternion128)
- T -> [N]T
- T -> matrix[R, C]T
- T -> #simd[N]T
- distinct proc <-> proc (same base types)
- distinct matrix <-> matrix (same base types)
- Subtypes through using or #subtype
- proc(^T) -> proc(^U) where T has a using or #subtype field of type U at offset 0 (see [subtype polymorphism](#))
- Untyped integers -> all numeric related types that can represent them without truncation
- Untyped floats -> all numeric related types that can represent them without truncation
- Untyped booleans -> all boolean related types
- Untyped rune -> all rune types

- Untyped strings -> all string types

## What to do if you get stuck

If you are unable to find the information you need on this overview page, the FAQ, the [wiki](#), the [Examples](#) (which contains demos for using the vendor libs, e.g. [Tetris in raylib](#)), or the [package documentation](#), then you will likely find it very helpful to explore your Odin source code directory, especially the files contained in the following five subdirectories: `examples\demo`, `tests`, `core`, `vendor`, and `misc\old_demos`.

The following list describes why each of these may be useful:

- The `examples\demo\demo.odin` file contains ~2600 lines of Odin code that demonstrate many of Odin's language features and provide commentary on some of them. This file contains many examples that cannot be found anywhere in the Overview nor in the other documentation.
- The `tests` folder provides unit tests for many of Odin's features and libraries, whereby reading these files you can gain a more nuanced view of exactly how things are intended or expected to behave. Tests are essentially automatically verified documentation. Tests often cover details that normal documentation doesn't even mention. Tests are also guaranteed to be up-to-date (if regularly run), unlike more informal documentation.
- The `core` and `vendor` folders contain the bindings and implementation code for Odin's standard library and various multimedia libraries respectively. **Even if you don't understand the implementation of something, reading the code can still greatly improve your chances of guessing what exactly it really does.** Odin is a straightforward and legible language with relatively little magic compared to other modern languages. Reading source code can enable you to work around missing documentation much more effectively. This is a very useful thing to keep in mind for working with any under-development language or library effectively.
- `misc\old_demos` may contain examples that are different from the main up-to-date `demo.odin` file contained in `examples\demo`. This may provide additional context for Odin's design and features. The other resources are probably better though.

Try doing a text search in one or more of these subdirectories for what you are looking for. Make sure the search is a proper full text search too, such as what `grep` (a popular 3rd party file search program) provides. Be careful that you specify your search correctly, especially if you use *regex*. Some default searches on some operating systems may miss content that exists in some files by not searching the actual file contents completely or at all, which can be misleading.

Odin's documentation may be sparse at times, but this sparsity can be worked around effectively by using the above listed resources. *Don't forget they exist.*

When working with any system that is still under-development these kinds of techniques are essential for maximizing your effectiveness. Many (perhaps most) real world companies, especially in intense industries such as game development and multimedia often don't have complete documentation and changes can happen rapidly.

Self-sufficiency in such a context requires a willingness to explore these kinds of resources proactively, rather than waiting for documentation that may be slow to come or may not fully cover what things actually do precisely enough.

Don't be afraid to simply experiment and try to deduce what something is really doing or what features may exist. You may be surprised how much you can still accomplish with just a little bit of patience and thoughtfulness!

Furthermore, even if Odin doesn't yet have a feature or library that you need, the fact that Odin has bindings to C's standard library (which are available in [the libc package](#) via `import "core:c/libc"`) means that any algorithms, example code, documentation, or books originally written for C can also be used to accomplish what you need in Odin. You can then optionally write Odin wrappers around that code to bridge the idiomatic gap better or else just use the C-like code directly. You can also bind to code from any libraries or languages that `foreign` can bind to (see [the foreign section](#) for more info).

Thus, when necessity calls for it, **the entire literature and community of C is still (in effect) available to you in Odin as well.** So, any time you are tempted to think "I can't implement this in Odin yet because there's not enough documentation" remind yourself that all (or almost all) of C is still available to you, but through a cleaner namespaced interface. Odin is also similar to C anyway, so direct translations of C to Odin (without even using the `libc` bindings) may still be relatively easy. In effect, any tutorial or book on C can also be thought of as an Odin resource too in that sense!

You can also make your experience navigating Odin code easier by installing a [syntax highlighter](#) and/or a [language server](#), which do exist for some editors (e.g. VS Code, Vim, Sublime Text, and Emacs) and work well enough to make using Odin comfortable. See [the list of text editors that have support for Odin code](#) for known examples.

Finally, you can ask for help on [the Odin forum](#) or from [the Odin community on Discord](#).

There's a collection of other [libraries created by the Odin community](#) that may also be helpful.

Good luck and have fun!