

Odin Demo Examples

Ginger Bill and the Odin contributors
Oct 14, 2026

Source: [demo.odin](#)

The original demonstration of the Odin in a single file, including numerous examples of concepts and features of the programming language.

```
#+vet !using-stmt !using-param
#+feature dynamic-literals using-stmt
package main

import "core:fmt"
import "core:mem"
import "core:os"
import "core:thread"
import "core:time"
import "core:reflect"
import "base:runtime"
import "base:intrinsics"
import "core:math/big"
import "core:math/rand"

/*
  Odin is a general-purpose programming language with distinct typing built
  for high performance, modern systems and data-oriented programming.

  Odin is the C alternative for the Joy of Programming.

  # Installing Odin
  Getting Started - https://odin-lang.org/docs/install/
  Instructions for downloading and install the Odin compiler and libraries.

  # Learning Odin
  Getting Started - https://odin-lang.org/docs/install/
  Getting Started with Odin. Downloading, installing, and getting your
  first program to compile and run.
  Overview of Odin - https://odin-lang.org/docs/overview/
  An overview of the Odin programming language and its features.
  Frequently Asked Questions (FAQ) - https://odin-lang.org/docs/faq/
  Answers to common questions about Odin.
  Packages - https://pkg.odin-lang.org/
  Documentation for all the official packages part of the
  core and vendor library collections.
  Nightly Builds - https://odin-lang.org/docs/nightly/
  Get the latest nightly builds of Odin.
  More Odin Examples - https://github.com/odin-lang/examples
  This repository contains examples of how certain things can be accomplished
  in idiomatic Odin, allowing you learn its semantics, as well as how to use
  parts of the core and vendor package collections.
*/

the_basics :: proc() {
  fmt.println("\n# the basics")

  { // The Basics

    // os.args holds the path to the current executable and any arguments passed to it.
    if len(os.args) == 1 {
      fmt.printf("Hellope from %v.\n", os.args[0])
    } else if len(os.args) > 2 {
      fmt.printf("%v, %v! from %v.\n", os.args[1], os.args[2], os.args[0])
    }

    // Lexical elements and literals
    // A comment

    my_integer_variable: int // A comment for documentaton

    // 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!
    */
    */

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

```

some_string := "This is a string"
_ = 'A' // unicode codepoint literal
_ = '\n'
_ = "C:\\Windows\\notepad.exe"
// Raw string literals are enclosed with single back ticks
_ = `C:\Windows\notepad.exe`

// The length of a string in bytes can be found using the built-in `len` procedure:
_ = len("Foo")
_ = len(some_string)

// 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, is an imaginary number literal: 2i (2 multiply the square root of -1).

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

// In Odin, if a numeric 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.

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

z: f64 // `z` is typed of type `f64` (64-bit floating point number)
z = 1 // `1` is an untyped integer literal which can be implicitly converted to `f64`
// No need for any suffixes or decimal places like in other languages
// (with the exception of negative zero, which must be given as `-0.0`)
// CONSTANTS JUST WORK!!!

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

// `:=` is the assignment operator

// You can assign multiple variables with it:
a, b := 1, "hello" // declares `a` and `b` and infers the types from the assignments
b, a = "byte", 0

// Note: `:=` is two tokens, `:` and `=`. The following are equivalent,
/*
i: int = 123
i:    = 123
i := 123
*/

// 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

_ = my_integer_variable
_ = x
}

control_flow :: proc() {
  fmt.println("\n# control flow")
  { // Control flow
    // For loop
    // Odin has only one loop statement, the `for` loop

    // Basic for loop
    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 fmt.print()

    // 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, an infinite loop is produced:
for {
    break
}

// Range-based for loop
// The basic for loop
for j := 0; j < 10; j += 1 {
    fmt.Println(j)
}
// can also be written
for j in 0..<10 {
    fmt.Println(j)
}
for j in 0..=9 {
    fmt.Println(j)
}

// Certain built-in types can be iterated over
some_string := "Hello, []"
for character in some_string { // Strings are assumed to be UTF-8
    fmt.Println(character)
}

some_array := []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 := []dynamicint{1, 4, 9}
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}
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.
// The following idiom is useful for iterating over a container in a by-reference manner:
for _, idx in some_slice {
    some_slice[idx] = (idx+1)*(idx+1)
}

// If statements
x := 123
if x >= 0 {
    fmt.Println("x is positive")
}

if y := -34; y < 0 {
    fmt.Println("y is negative")
}

if y := 123; y < 0 {
    fmt.Println("y is negative")
} else if y == 0 {
    fmt.Println("y is zero")
} else {
    fmt.Println("y 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.

#partial switch arch := ODIN_ARCH; arch {
case .i386:
    fmt.println("32-bit")
case .amd64:
    fmt.println("64-bit")
case: // default
    fmt.println("Unsupported architecture")
}

// Odin's `switch` is like 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.
one_angry_dwarf :: proc() -> int {
    fmt.println("one_angry_dwarf was called")
    return 1
}

switch j := 0; j {
case 0:
case one_angry_dwarf():
}

// 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 := 'j'; 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")
}
}

{ // 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:
{
    x := 123
    defer fmt.println(x)
    {
        defer x = 4
        x = 2
    }
    fmt.println(x)

    x = 234
}

// You can defer an entire block too:
{
    bar :: proc() {}

    defer {
        fmt.println("1")
        fmt.println("2")
    }

    cond := false
    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.

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

{ // 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("Unknown architecture")
}
// The when statement is very useful for writing platform specific code.
// This is akin to the #if construct in C's preprocessor however, in Odin,
// it is type checked.
}

{ // Branch statements
cond, cond1, cond2 := false, false, false
one_step :: proc() { fmt.Println("one_step") }
beyond :: proc() { fmt.Println("beyond") }

// Break statement
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
    }
}

// Continue statement
for cond {
    if cond2 {
        continue
    }
    fmt.Println("Hellope")
}

// Fallthrough statement

// Odin's switch is like 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 := 0; i {
case 0:
    one_step()
    fallthrough
case 1:
    beyond()
}
}

named_proc_return_parameters :: proc() {
    fmt.Println("\n# named proc return parameters")
}

```

```

foo0 :: proc() -> int {
    return 123
}
foo1 :: proc() -> (a: int) {
    a = 123
    return
}
foo2 :: proc() -> (a, b: int) {
    // Named return values act like variables within the scope
    a = 321
    b = 567
    return b, a
}
fmt.println("foo0 =", foo0()) // 123
fmt.println("foo1 =", foo1()) // 123
fmt.println("foo2 =", foo2()) // 567 321
}

variadic_procedures :: proc() {
    fmt.println("\n# variadic procedures")
    sum :: proc(nums: ..int, init_value:= 0) -> (result: int) {
        result = init_value
        for n in nums {
            result += n
        }
        return
    }
    fmt.println("sum(()) =", sum())
    fmt.println("sum(1, 2) =", sum(1, 2))
    fmt.println("sum(1, 2, 3, 4, 5) =", sum(1, 2, 3, 4, 5))
    fmt.println("sum(1, 2, 3, 4, 5, init_value = 5) =", sum(1, 2, 3, 4, 5, init_value = 5))

    // pass a slice as varargs
    odds := []int{1, 3, 5}
    fmt.println("odds =", odds)
    fmt.println("sum(..odds) =", sum(..odds))
    fmt.println("sum(..odds, init_value = 5) =", sum(..odds, init_value = 5))
}

explicit_procedure_overloading :: proc() {
    fmt.println("\n# explicit procedure overloading")

    add_ints :: proc(a, b: int) -> int {
        x := a + b
        fmt.println("add_ints", x)
        return x
    }
    add_floats :: proc(a, b: f32) -> f32 {
        x := a + b
        fmt.println("add_floats", x)
        return x
    }
    add_numbers :: proc(a: int, b: f32, c: u8) -> int {
        x := int(a) + int(b) + int(c)
        fmt.println("add_numbers", x)
        return x
    }
}

add :: proc{add_ints, add_floats, add_numbers}

add(int(1), int(2))
add(f32(1), f32(2))
add(int(1), f32(2), u8(3))

add(1, 2) // untyped ints coerce to int tighter than f32
add(1.0, 2.0) // untyped floats coerce to f32 tighter than int
add(1, 2, 3) // three parameters

// Ambiguous answers
// add(1.0, 2)
// add(1, 2.0)
}

struct_type :: proc() {
    fmt.println("\n# struct type")
    // 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 abstract the ability
// to not explicitly dereference the pointer. This is very useful when
// refactoring code to use a pointer rather than a value, and vice versa.
}
{
  // 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 tagged with different memory layout and alignment requirements:

  a :: struct #align(4) {} // align to 4 bytes
  b :: struct #packed {} // remove padding between fields
  c :: struct #raw_union {} // all fields share the same offset (0). This is the same as C's union
}
}

union_type :: proc() {
  fmt.println("\n# union type")
  {
    val: union{int, bool}
    val = 137
    if i, ok := val.(int); ok {
      fmt.println(i)
    }
    val = true
    fmt.println(val)

    val = nil

    switch v in val {
    case int: fmt.println("int", v)
    case bool: fmt.println("bool", v)
    case: fmt.println("nil")
    }
  }
  {
    // There is a duality between `any` and `union`
    // An `any` has a pointer to the data and allows for any type (open)
    // A `union` has as binary blob to store the data and allows only certain types (closed)
    // The following code is with `any` but has the same syntax
    val: any
    val = 137
    if i, ok := val.(int); ok {
      fmt.println(i)
    }
    val = true
    fmt.println(val)

    val = nil

    switch v in val {
    case int: fmt.println("int", v)
    case bool: fmt.println("bool", v)
    case: fmt.println("nil")
    }
  }
}

Vector3 :: distinct [3]f32
Quaternion :: distinct quaternion128

// More realistic examples
{
  // NOTE(bill): For the above basic examples, you may not have any
  // particular use for it. However, my main use for them is not for these
  // simple cases. My main use is for hierarchical types. Many prefer
  // subtyping, embedding the base data into the derived types. Below is
  // an example of this for a basic game Entity.

  Entity :: struct {
    id: u64,
    name: string,
    position: Vector3,
    orientation: Quaternion,

    derived: any,
  }
}

```

```

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

Monster :: struct {
    using entity: Entity,
    is_robot:    bool,
    is_zombie:   bool,
}

// See `parametric_polymorphism` procedure for details
new_entity :: proc($T: typeid) -> ^Entity {
    t := new(T)
    t.derived = t^
    return t
}

entity := new_entity(Monster)

switch e in entity.derived {
case Frog:
    fmt.println("Ribbit")
case Monster:
    if e.is_robot { fmt.println("Robotic") }
    if e.is_zombie { fmt.println("Grrrr!") }
    fmt.println("I'm a monster")
}
}

{
// NOTE(bill): A union can be used to achieve something similar. Instead
// of embedding the base data into the derived types, the derived data
// is embedded into the base type. Below is the same example of the
// basic game Entity but using an union.

Entity :: struct {
    id:        u64,
    name:      string,
    position:   Vector3,
    orientation: Quaternion,

    derived: union {Frog, Monster},
}

Frog :: struct {
    using entity: ^Entity,
    jump_height: f32,
}

Monster :: struct {
    using entity: ^Entity,
    is_robot:    bool,
    is_zombie:   bool,
}

// See `parametric_polymorphism` procedure for details
new_entity :: proc($T: typeid) -> ^Entity {
    t := new(Entity)
    t.derived = T{entity = t}
    return t
}

entity := new_entity(Monster)

switch e in entity.derived {
case Frog:
    fmt.println("Ribbit")
case Monster:
    if e.is_robot { fmt.println("Robotic") }
    if e.is_zombie { fmt.println("Grrrr!") }
}

// NOTE(bill): As you can see, the usage code has not changed, only its
// memory layout. Both approaches have their own advantages but they can
// be used together to achieve different results. The subtyping approach
// can allow for a greater control of the memory layout and memory
// allocation, e.g. storing the derivatives together. However, this is
// also its disadvantage. You must either preallocate arrays for each
// derivative separation (which can be easily missed) or preallocate a
// bunch of "raw" memory; determining the maximum size of the derived
// types would require the aid of metaprogramming. Unions solve this
// particular problem as the data is stored with the base data.
// Therefore, it is possible to preallocate, e.g. [100]Entity.

// It should be noted that the union approach can have the same memory
// layout as the any and with the same type restrictions by using a
// pointer type for the derivatives.

/*
Entity :: struct {
    ...
    derived: union{^Frog, ^Monster},

```

```

    }

    Frog :: struct {
        using entity: Entity,
        ...
    }
    Monster :: struct {
        using entity: Entity,
        ...
    }

    new_entity :: proc(T: type) -> ^Entity {
        t := new(T)
        t.derived = t
        return t
    }
}
*/
}
}

using_statement :: proc() {
    // IMPORTANT NOTE: `using` as a statement is an opt-in feature which can be abled
    // by adding `##feature using-stmt` to be beginning of the file
    //
    // `using` as a struct field modifier remains available always

    fmt.println("\n# using statement")
    // using can be used to bring entities declared in a scope/namespace
    // into the current scope. This can be applied to import names, struct
    // fields, procedure fields, and struct values.

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

        // It can be used like this:
        foo0 :: proc(entity: ^Entity) {
            fmt.println(entity.position.x, entity.position.y, entity.position.z)
        }

        // The entity members can be brought into the procedure scope by using it:
        foo1 :: proc(entity: ^Entity) {
            using entity
            fmt.println(position.x, position.y, position.z)
        }

        // The using can be applied to the parameter directly:
        foo2 :: proc(using entity: ^Entity) {
            fmt.println(position.x, position.y, position.z)
        }

        // It can also be applied to sub-fields:
        foo3 :: proc(entity: ^Entity) {
            using entity.position
            fmt.println(x, y, z)
        }
    }
}

{
    // We can also apply the using statement to the struct fields directly,
    // making all the fields of position appear as if they on Entity itself:
    Entity :: struct {
        using position: Vector3,
        orientation: quaternion128,
    }
    foo :: proc(entity: ^Entity) {
        fmt.println(entity.x, entity.y, entity.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:

Colour :: struct {r, g, b, a: u8}
Frog :: struct {
    ribbit_volume: f32,
    using entity: Entity,
    colour: Colour,
}

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

// 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.

```

```

}
}

implicit_context_system :: proc() {
    fmt.println("\n# 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 he 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 is
    // not possible.

    c := context // copy the current scope's context

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

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

    what_a_fool_believes :: 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()`
        assert(context.user_index == 123)

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

        _ = c
    }

    my_custom_allocator :: mem.nil_allocator
    _ = c

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

    // To see what the implicit context value contains, please see the following
    // definition in package runtime.
}

parametric_polymorphism :: proc() {
    fmt.println("\n# parametric polymorphism")

    print_value :: proc(value: $T) {
        fmt.printf("print_value: %T %v\n", value, value)
    }

    v1: int    = 1
    v2: f32    = 2.1
    v3: f64    = 3.14
    v4: string = "message"

    print_value(v1)
    print_value(v2)
    print_value(v3)
    print_value(v4)

    fmt.println()

    add :: proc(p, q: $T) -> T {
        x: T = p + q
        return x
    }

    a := add(3, 4)
    fmt.printf("a: %T = %v\n", a, a)

    b := add(3.2, 4.3)
    fmt.printf("b: %T = %v\n", b, b)

    // This is how `new` is implemented
    alloc_type :: proc($T: typeid) -> ^T {
        t := cast(^T)mem.alloc(size_of(T), align_of(T))
        t^ = T{} // Use default initialization value
        return t
    }
}

```

```

copy_slice :: proc(dst, src: []$T) -> int {
  n := min(len(dst), len(src))
  if n > 0 {
    mem.copy(&dst[0], &src[0], n*size_of(T))
  }
  return n
}

double_params :: proc(a: $A, b: $B) -> A {
  return a + A(b)
}

fmt.println(double_params(12, 1.345))

{ // Polymorphic Types and Type Specialization
  Table_Slot :: struct($Key, $Value: typeid) {
    occupied: bool,
    hash:     u32,
    key:      Key,
    value:    Value,
  }
  TABLE_SIZE_MIN :: 32
  Table :: struct($Key, $Value: typeid) {
    count:    int,
    allocator: mem.Allocator,
    slots:    []Table_Slot(Key, Value),
  }

  // Only allow types that are specializations of a (polymorphic) slice
  make_slice :: proc($T: typeid/[]$E, len: int) -> T {
    return make(T, len)
  }

  // Only allow types that are specializations of `Table`
  allocate :: proc(table: ^$T/Table, capacity: int) {
    c := context
    if table.allocator.procedure != nil {
      c.allocator = table.allocator
    }
    context = c

    table.slots = make_slice(type_of(table.slots), max(capacity, TABLE_SIZE_MIN))
  }

  expand :: proc(table: ^$T/Table) {
    c := context
    if table.allocator.procedure != nil {
      c.allocator = table.allocator
    }
    context = c

    old_slots := table.slots
    defer delete(old_slots)

    cap := max(2*len(table.slots), TABLE_SIZE_MIN)
    allocate(table, cap)

    for s in old_slots {
      if s.occupied {
        put(table, s.key, s.value)
      }
    }
  }

  // Polymorphic determination of a polymorphic struct
  // put :: proc(table: ^$T/Table, key: T.Key, value: T.Value) {
  put :: proc(table: ^Table($Key, $Value), key: Key, value: Value) {
    hash := get_hash(key) // Ad-hoc method which would fail in a different scope
    index := find_index(table, key, hash)
    if index < 0 {
      if f64(table.count) >= 0.75*f64(len(table.slots)) {
        expand(table)
      }
      assert(table.count <= len(table.slots))

      index = int(hash % u32(len(table.slots)))

      for table.slots[index].occupied {
        if index += 1; index >= len(table.slots) {
          index = 0
        }
      }

      table.count += 1
    }

    slot := &table.slots[index]
    slot.occupied = true
    slot.hash     = hash
    slot.key      = key
    slot.value    = value
  }
}

```

```

// find :: proc(table: ^T/Table, key: T.Key) -> (T.Value, bool) {
find :: proc(table: ^Table($Key, $Value), key: Key) -> (Value, bool) {
  hash := get_hash(key)
  index := find_index(table, key, hash)
  if index < 0 {
    return Value{}, false
  }
  return table.slots[index].value, true
}

find_index :: proc(table: ^Table($Key, $Value), key: Key, hash: u32) -> int {
  if len(table.slots) <= 0 {
    return -1
  }

  index := int(hash % u32(len(table.slots)))
  for table.slots[index].occupied {
    if table.slots[index].hash == hash {
      if table.slots[index].key == key {
        return index
      }
    }

    if index += 1; index >= len(table.slots) {
      index = 0
    }
  }

  return -1
}

get_hash :: proc(s: string) -> u32 { // fnv32a
  h: u32 = 0x811c9dc5
  for i in 0..

```

```

    assert(v == T(i*i))
}

// Matrix multiplication
mul :: proc(a: [M][N]$T, b: [N][P]$T) -> (c: [M][P]$T) {
    for i in 0..

```

```

        fmt.printf("`s`: iteration %d\n", prefix_table[index], iteration)

        print_mutex = false

        time.sleep(1 * time.Millisecond)
    }
}

N :: 3

pool: thread.Pool
thread.pool_init(&pool, allocator=context.allocator, thread_count=N)
defer thread.pool_destroy(&pool)

for i in 0..<30 {
    // be mindful of the allocator used for tasks. The allocator needs to be thread safe, or be owned by the task for
    exclusive use
    thread.pool_add_task(&pool, allocator=context.allocator, procedure=task_proc, data=nil, user_index=i)
}

thread.pool_start(&pool)
thread.pool_finish(&pool)
}
}

array_programming :: proc() {
    fmt.println("\n# array programming")
    {
        a := [3]f32{1, 2, 3}
        b := [3]f32{5, 6, 7}
        c := a * b
        d := a + b
        e := 1 + (c - d) / 2
        fmt.printf("%.1f\n", e) // [0.5, 3.0, 6.5]
    }

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

        c := swizzle(a, 0, 0)
        assert(c == [2]f32{1, 1})
        assert(c == 1)
    }

    {
        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))
    }
}

map_type :: proc() {
    fmt.println("\n# map type")

    m := make(map[string]int)
    defer delete(m)

    m["Bob"] = 2
    m["Ted"] = 5
    fmt.println(m["Bob"])

    delete_key(&m, "Ted")

    // If an element of a key does not exist, the zero value of the
    // element will be returned. To check to see if an element exists
    // can be done in two ways:
    elem, ok := m["Bob"]

```

```

    exists := "Bob" in m
    _, _ = elem, ok
    _ = exists
}

implicit_selector_expression :: proc() {
    fmt.println("\n# implicit selector expression")

    Foo :: enum {A, B, C}

    f: Foo
    f = Foo.A
    f = .A

    BAR :: bit_set[Foo]{.B, .C}

    switch f {
    case .A:
        fmt.println("HITHER")
    case .B:
        fmt.println("NEVER")
    case .C:
        fmt.println("FOREVER")
    }

    my_map := make(map[Foo]int)
    defer delete(my_map)

    my_map[.A] = 123
    my_map[Foo.B] = 345

    fmt.println(my_map[.A] + my_map[Foo.B] + my_map[.C])
}

partial_switch :: proc() {
    fmt.println("\n# partial_switch")
    { // enum
        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")
        }
    }
    { // union
        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")
        }
    }
}

cstring_example :: proc() {
    fmt.println("\n# cstring_example")

    W :: "Hellope"
    X :: cstring(W)
    Y :: string(X)

    w := W
    _ = w
    x: cstring = X
    y: string = Y
    z := string(x)
    fmt.println(x, y, z)
    fmt.println(len(x), len(y), len(z))
    fmt.println(len(W), len(X), len(Y))
    // IMPORTANT NOTE for cstring variables
    // len(cstring) is 0(N)
    // cast(string)cstring is 0(N)
}

```

```

bit_set_type :: proc() {
    fmt.println("\n# bit_set type")

    {
        Day :: enum {
            Sunday,
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday,
        }

        Days :: distinct bit_set[Day]
        WEEKEND :: Days{.Sunday, .Saturday}

        d: Days
        d = {.Sunday, .Monday}
        e := d + WEEKEND
        e += {.Monday}
        fmt.println(d, e)

        ok := .Saturday in e // `in` is only allowed for `map` and `bit_set` types
        fmt.println(ok)
        if .Saturday in e {
            fmt.println("Saturday in", e)
        }
        X :: .Saturday in WEEKEND // Constant evaluation
        fmt.println(X)
        fmt.println("Cardinality:", card(e))
    }
    {
        x: bit_set['A'..'Z']
        #assert(size_of(x) == size_of(u32))
        y: bit_set[0..=8; u16]
        fmt.println(typeid_of(type_of(x))) // bit_set[A..=Z]
        fmt.println(typeid_of(type_of(y))) // bit_set[0..=8; u16]

        x += {'F'}
        assert('F' in x)
        x -= {'F'}
        assert('F' not_in x)

        y += {1, 4, 2}
        assert(2 in y)
    }
    {
        Letters :: bit_set['A'..'Z']
        a := Letters{'A', 'B'}
        b := Letters{'A', 'B', 'C', 'D', 'F'}
        c := Letters{'A', 'B'}

        assert(a <= b) // 'a' is a subset of 'b'
        assert(b >= a) // 'b' is a superset of 'a'
        assert(a < b) // 'a' is a strict subset of 'b'
        assert(b > a) // 'b' is a strict superset of 'a'

        assert(!(a < c)) // 'a' is a not strict subset of 'c'
        assert(!(c > a)) // 'c' is a not strict superset of 'a'
    }
}

deferred_procedure_associations :: proc() {
    fmt.println("\n# deferred procedure associations")

    @(deferred_out=closure)
    open :: proc(s: string) -> bool {
        fmt.println(s)
        return true
    }

    closure :: proc(ok: bool) {
        fmt.println("Goodbye?", ok)
    }

    if open("Welcome") {
        fmt.println("Something in the middle, mate.")
    }
}

reflection :: proc() {
    fmt.println("\n# reflection")

    Foo :: struct {
        x: int `tag1`,
        y: string `json:"y_field"`,
        z: bool, // no tag
    }

    id := typeid_of(Foo)
    names := reflect.struct_field_names(id)
    types := reflect.struct_field_types(id)
    tags := reflect.struct_field_tags(id)

```

```

assert(len(names) == len(types) && len(names) == len(tags))

fmt.println("Foo :: struct {")
for tag, i in tags {
    name, type := names[i], types[i]
    if tag != "" {
        fmt.printf("\t%s: %T `s`,\n", name, type, tag)
    } else {
        fmt.printf("\t%s: %T,\n", name, type)
    }
}
fmt.println("}")

for tag, i in tags {
    if val, ok := reflect.struct_tag_lookup(tag, "json"); ok {
        fmt.printf("json: %s -> %s\n", names[i], val)
    }
}
}

quaternions :: proc() {
    // Not just an April Fool's Joke any more, but a fully working thing!
    fmt.println("\n# quaternions")

    { // Quaternion operations
        q := 1 + 2i + 3j + 4k
        r := quaternion(real=5, imag=6, jmag=7, kmag=8)
        t := q * r
        fmt.printf("(%v) * (%v) = %v\n", q, r, t)
        v := q / r
        fmt.printf("(%v) / (%v) = %v\n", q, r, v)
        u := q + r
        fmt.printf("(%v) + (%v) = %v\n", q, r, u)
        s := q - r
        fmt.printf("(%v) - (%v) = %v\n", q, r, s)
    }
    { // The quaternion types
        q128: quaternion128 // 4xf32
        q256: quaternion256 // 4xf64
        q128 = quaternion(w=1, x=0, y=0, z=0)
        q256 = 1 // quaternion(x=0, y=0, z=0, w=1)

        // NOTE: The internal memory layout of a quaternion is xyzw
    }
    { // Built-in procedures
        q := 1 + 2i + 3j + 4k
        fmt.println("q =", q)
        fmt.println("real(q) =", real(q))
        fmt.println("imag(q) =", imag(q))
        fmt.println("jmag(q) =", jmag(q))
        fmt.println("kmag(q) =", kmag(q))
        fmt.println("conj(q) =", conj(q))
        fmt.println("abs(q) =", abs(q))
    }
    { // Conversion of a complex type to a quaternion type
        c := 1 + 2i
        q := quaternion256(c)
        fmt.println(c)
        fmt.println(q)
    }
    { // Memory layout of Quaternions
        q := 1 + 2i + 3j + 4k
        a := transmute([4]f64)q
        fmt.println("Quaternion memory layout: xyzw/(ijkr)")
        fmt.println(q) // 1.000+2.000i+3.000j+4.000k
        fmt.println(a) // [2.000, 3.000, 4.000, 1.000]
    }
}

unroll_for_statement :: proc() {
    fmt.println("\n#unroll for' statements")

    // '#unroll for' works the same as if the 'inline' prefix did not
    // exist but these ranged loops are explicitly unrolled which can
    // be very very useful for certain optimizations

    fmt.println("Ranges")
    #unroll for x, i in 1..<4 {
        fmt.println(x, i)
    }

    fmt.println("Strings")
    #unroll for r, i in "Hello, []" {
        fmt.println(r, i)
    }

    fmt.println("Arrays")
    #unroll for elem, idx in ([4]int{1, 4, 9, 16}) {
        fmt.println(elem, idx)
    }
}

```

```

Foo_Enum :: enum {
  A = 1,
  B,
  C = 6,
  D,
}
fmt.println("Enum types")
#unroll for elem, idx in Foo_Enum {
  fmt.println(elem, idx)
}
}

where_clauses :: proc() {
  fmt.println("\n#procedure 'where' clauses")

  { // Sanity checks
    simple_sanity_check :: proc(x: [2]int)
      where len(x) > 1,
            type_of(x) == [2]int {
        fmt.println(x)
      }
  }
  { // Parametric polymorphism checks
    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))
  }

  { // Procedure groups usage
    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)
  }

  { // 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))
  }
}

when ODIN_OS == .Windows {
  foreign import kernel32 "system:kernel32.lib"
}

```

```

foreign_system :: proc() {
  fmt.println("\n#foreign system")
  when ODIN_OS == .Windows {
    // It is sometimes necessarily 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.

    // This foreign import declaration will create a
    // "foreign import name" which can then be used to associate
    // entities within a foreign block.

    foreign kernel32 {
      ExitProcess :: proc "stdcall" (exit_code: u32) ---
    }

    // Foreign procedure declarations have the cdecl/c calling
    // convention by default unless specified otherwise. Due to
    // foreign procedures do not have a body declared within this
    // code, you need 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 ---
    }

    // Example using the link_prefix attribute
    @(default_calling_convention = "std")
    @(link_prefix = "Get")
    foreign kernel32 {
      GetLastError :: proc() -> i32 ---
    }
  }
}

ranged_fields_for_array_compound_literals :: proc() {
  fmt.println("\n#ranged fields for array compound literals")
  { // Normal Array Literal
    foo := [?]int{1, 4, 9, 16}
    fmt.println(foo)
  }
  { // Indexed
    foo := [?]int{
      3 = 16,
      1 = 4,
      2 = 9,
      0 = 1,
    }
    fmt.println(foo)
  }
  { // Ranges
    i := 2
    foo := [?]int {
      0 = 123,
      5..<9 = 54,
      10..<16 = i*3 + (i-1)*2,
    }
    #assert(len(foo) == 16)
    fmt.println(foo) // [123, 0, 0, 0, 0, 54, 54, 54, 54, 54, 8, 8, 8, 8, 8]
  }
  { // Slice and Dynamic Array support
    i := 2
    foo_slice := [int] {
      0 = 123,
      5..<9 = 54,
      10..<16 = i*3 + (i-1)*2,
    }
    assert(len(foo_slice) == 16)
    fmt.println(foo_slice) // [123, 0, 0, 0, 0, 54, 54, 54, 54, 54, 8, 8, 8, 8, 8]

    foo_dynamic_array := [dynamic]int {
      0 = 123,
      5..<9 = 54,
      10..<16 = i*3 + (i-1)*2,
    }
    assert(len(foo_dynamic_array) == 16)
    fmt.println(foo_dynamic_array) // [123, 0, 0, 0, 0, 54, 54, 54, 54, 54, 8, 8, 8, 8, 8]
  }
}

deprecated_attribute :: proc() {
  @(deprecated="Use foo_v2 instead")
  foo_v1 :: proc(x: int) {
    fmt.println("foo_v1")
  }
  foo_v2 :: proc(x: int) {
    fmt.println("foo_v2")
  }
}

// NOTE: Uncomment to see the warning messages

```

```

} // foo_v1(1)
}

range_statements_with_multiple_return_values :: proc() {
  fmt.println("\n#range statements with multiple return values")
  My_Iterator :: struct {
    index: int,
    data: []i32,
  }
  make_my_iterator :: proc(data: []i32) -> My_Iterator {
    return My_Iterator{data = data}
  }
  my_iterator :: proc(it: ^My_Iterator) -> (val: i32, idx: int, cond: bool) {
    if cond = it.index < len(it.data); cond {
      val = it.data[it.index]
      idx = it.index
      it.index += 1
    }
    return
  }
}

data := make([]i32, 6)
for _, i in data {
  data[i] = i32(i*i)
}

{ // Manual Style
  it := make_my_iterator(data)
  for {
    val, _, cond := my_iterator(&it)
    if !cond {
      break
    }
    fmt.println(val)
  }
}
{ // or_break
  it := make_my_iterator(data)
  loop: for {
    val, _ := my_iterator(&it) or_break loop
    fmt.println(val)
  }
}
{ // first value
  it := make_my_iterator(data)
  for val in my_iterator(&it) {
    fmt.println(val)
  }
}
{ // first and second value
  it := make_my_iterator(data)
  for val, idx in my_iterator(&it) {
    fmt.println(val, idx)
  }
}
}

soa_struct_layout :: proc() {
  fmt.println("\n#SOA Struct Layout")

  {
    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 <= 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 Slices
// Vector3 :: struct {x, y, z: f32}
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)

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[:])
}
{ // soa_zip and soa_unzip
fmt.println("\nssoa_zip and soa_unzip")

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

// produce an #soa slice 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)
}

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

constant_literal_expressions :: proc() {
fmt.println("\n#constant literal expressions")

Bar :: struct {x, y: f32}
Foo :: struct {a, b: int, using c: Bar}

FOO_CONST :: Foo{b = 2, a = 1, c = {3, 4}}

```

```

fmt.println(FOO_CONST.a)
fmt.println(FOO_CONST.b)
fmt.println(FOO_CONST.c)
fmt.println(FOO_CONST.c.x)
fmt.println(FOO_CONST.c.y)
fmt.println(FOO_CONST.x) // using works as expected
fmt.println(FOO_CONST.y)

fmt.println("-----")

ARRAY_CONST :: [3]int{1 = 4, 2 = 9, 0 = 1}

fmt.println(ARRAY_CONST[0])
fmt.println(ARRAY_CONST[1])
fmt.println(ARRAY_CONST[2])

fmt.println("-----")

FOO_ARRAY_DEFAULTS :: [3]Foo{ {}, {}, {} }
fmt.println(FOO_ARRAY_DEFAULTS[2].x)

fmt.println("-----")

Baz :: enum{A=5, B, C, D}
ENUM_ARRAY_CONST :: [Baz]int{.A == .C = 1, .D = 16}

fmt.println(ENUM_ARRAY_CONST[.A])
fmt.println(ENUM_ARRAY_CONST[.B])
fmt.println(ENUM_ARRAY_CONST[.C])
fmt.println(ENUM_ARRAY_CONST[.D])

fmt.println("-----")

Sparse_Baz :: enum{A=5, B, C, D=16}
#assert(len(Sparse_Baz) < len(#sparse[Sparse_Baz]int))
SPARSE_ENUM_ARRAY_CONST :: #sparse[Sparse_Baz]int{.A == .C = 1, .D = 16}

fmt.println(SPARSE_ENUM_ARRAY_CONST[.A])
fmt.println(SPARSE_ENUM_ARRAY_CONST[.B])
fmt.println(SPARSE_ENUM_ARRAY_CONST[.C])
fmt.println(SPARSE_ENUM_ARRAY_CONST[.D])

fmt.println("-----")

STRING_CONST :: "Hello!"

fmt.println(STRING_CONST[0])
fmt.println(STRING_CONST[2])
fmt.println(STRING_CONST[3])

fmt.println(STRING_CONST[0:5])
fmt.println(STRING_CONST[3:][4])
}

union_maybe :: proc() {
  fmt.println("\n#union based maybe")

  // NOTE: This is already built-in, and this is just a reimplementaiton to explain the behaviour
  Maybe :: union($T: typeid) {T}

  i: Maybe(u8)
  p: Maybe(^u8) // No tag is stored for pointers, nil is the sentinel value

  // Tag size will be as small as needed for the number of variants
  #assert(size_of(i) == size_of(u8) + size_of(u8))
  // No need to store a tag here, the `nil` state is shared with the variant's `nil`
  #assert(size_of(p) == size_of(^u8))

  i = 123
  x := i.?
  y, y_ok := p.?
  p = &x
  z, z_ok := p.?

  fmt.println(i, p)
  fmt.println(x, &x)
  fmt.println(y, y_ok)
  fmt.println(z, z_ok)
}

dummy_procedure :: proc() {
  fmt.println("dummy_procedure")
}

explicit_context_definition :: proc "c" () {
  // Try commenting the following statement out below
  context = runtime.default_context()

  fmt.println("\n#explicit context definition")
  dummy_procedure()
}

```

```

or_else_operator :: proc() {
  fmt.println("\n#or_else'")
  {
    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 :: proc() {
  fmt.println("\n#or_return'")
  // 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 'false', and if so,
  // set the end return value to value if possible. 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 parameters 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 multiple parameters (2+),
    // then all the parameters must be named so that the remaining parameters
    // 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
}

defer if err != nil {
    fmt.Println("Error in", #procedure, ":", err)
}

n = 123
return
}

foo_1()
foo_2()
}

or_break_and_or_continue_operators :: proc() {
    fmt.Println("\n#'or_break' and 'or_continue'")
    // The concept of 'or_break' and 'or_continue' is very similar to that of 'or_return'.
    // The difference is that unlike 'or_return', the value does not get returned from
    // the current procedure but rather discarded if it is 'false' or not 'nil', and then
    // the specified branch (i.e. break or continue).
    // The or branch expression can be labelled if a specific statement needs to be used.

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

    caller_1 :: proc() -> Error {
        return .Something_Bad
    }

    caller_2 :: proc() -> (int, Error) {
        return 123, .Something_Worse
    }
    caller_3 :: proc() -> (int, int, Error) {
        return 123, 345, .None
    }

    for { // common approach
        err := caller_1()
        if err != nil {
            break
        }
    }
    for { // or_break approach
        caller_1() or_break
    }

    for { // or_break approach with multiple values
        n := caller_2() or_break
        _ = n
    }

    loop: for { // or_break approach with named label
        n := caller_2() or_break loop
        _ = n
    }

    for { // or_continue
        x, y := caller_3() or_continue
        _, _ = x, y

        break
    }

    continue_loop: for { // or_continue with named label
        x, y := caller_3() or_continue continue_loop
        _, _ = x, y

        break
    }
}

```

```

}

arbitrary_precision_mathematics :: proc() {
    fmt.println("\n# core:math/big")

    print_bigint :: proc(name: string, a: ^big.Int, base := i8(10), print_name := true, newline := true, print_extra_info :=
true) {
        big.assert_if_nil(a)

        as, err := big.itoa(a, base)
        defer delete(as)

        cb := big.internal_count_bits(a)
        if print_name {
            fmt.print(name)
        }
        if err != nil {
            fmt.printf(" (Error: %v) ", err)
        }
        fmt.printf(as)
        if print_extra_info {
            fmt.printf(" (base: %v, bits: %v, digits: %v)", base, cb, a.used)
        }
        if newline {
            fmt.println()
        }
    }
}

a, b, c, d, e, f, res := &big.Int{}, &big.Int{}, &big.Int{}, &big.Int{}, &big.Int{}, &big.Int{}, &big.Int{}
defer big.destroy(a, b, c, d, e, f, res)

// Set the context RNG to something that does not require
// cryptographic entropy (not supported on all targets).
context.random_generator = rand.xoshiro256_random_generator()

// How many bits should the random prime be?
bits := 64
// Number of Rabin-Miller trials, -1 for automatic.
trials := -1

// Default prime generation flags
flags := big.Primalty_Flags{}

err := big.internal_random_prime(a, bits, trials, flags)
if err != nil {
    fmt.printf("Error %v while generating random prime.\n", err)
} else {
    print_bigint("Random Prime A: ", a, 10)
    fmt.printf("Random number iterations until prime found: %v\n", big.RANDOM_PRIME_ITERATIONS_USED)
}

// If we want to pack this Int into a buffer of u32, how many do we need?
count := big.internal_int_pack_count(a, u32)
buf := make([]u32, count)
defer delete(buf)

written: int
written, err = big.internal_int_pack(a, buf)
fmt.printf("\nPacked into u32 buf: %v | err: %v | written: %v\n", buf, err, written)

// If we want to pack this Int into a buffer of bytes of which only the bottom 6 bits are used, how many do we need?
nails := 2

count = big.internal_int_pack_count(a, u8, nails)
byte_buf := make([]u8, count)
defer delete(byte_buf)

written, err = big.internal_int_pack(a, byte_buf, nails)
fmt.printf("\nPacked into buf of 6-bit bytes: %v | err: %v | written: %v\n", byte_buf, err, written)

// Pick another random big Int, not necessarily prime.
err = big.random(b, 2048)
print_bigint("\n2048 bit random number: ", b)

// Calculate GCD + LCM in one fell swoop
big.gcd_lcm(c, d, a, b)

print_bigint("\nGCD of random prime A and random number B: ", c)
print_bigint("\nLCM of random prime A and random number B (in base 36): ", d, 36)
}

matrix_type :: proc() {
    fmt.println("\n# 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
  assert(elem == -6)

  // 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.println("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
      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 := intrinsics.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 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", intrinsics.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: the 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 different alignment requirement)
// Column-major is used in order to utilize (SIMD) vector instructions effectively on
// modern hardware, if possible.
//
// Unlike normal arrays, matrices try to maximize alignment to allow for the (SIMD) vectorization
// properties whilst keeping zero padding (either between columns 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 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 16 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 minimizing the possibility of stack overflows.

// 'intrinsics' Procedures (Compiler Level)
// 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
// Example:

```

```

// m := matrix[2, 2]f32{
//   x0, x1,
//   y0, y1,
// }
// array: [4]f32 = matrix_flatten(m)
// assert(array == {x0, y0, x1, y1})
// conj(x)
// conjugates the elements of a matrix for complex element types only

// Procedures in "core:math/linalg" and related (Runtime Level) (all square matrix procedures)
// determinant(m)
// adjugate(m)
// inverse(m)
// inverse_transpose(m)
// hermitian_adjoint(m)
// trace(m)
// matrix_minor(m)
}

bit_field_type :: proc() {
  fmt.println("\n# bit_field type")
  // A `bit_field` is a record type in Odin that is akin to a bit-packed struct.
  // IMPORTANT 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 {
      x: i32 | 3, // backing type must be an integer or array of integers
      y: u16 | 2 + 3, // signed integers will be signed extended on use
      z: My_Enum | SOME_CONSTANT, // general expressions
      w: bool | 2 when SOME_CONSTANT > 10 else 1, // ability to define the bit-width elsewhere
    }

    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

    My_Enum :: enum u8 {A, B, C, D}
    SOME_CONSTANT :: 7
  }

  {
    // 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.

    Bar :: bit_field u32 {}
    Baz :: bit_field [4]u8 {}
  }

  // IMPORTANT NOTES:
  // * If _all_ of the fields in a bit_field are 1-bit in size and they are all booleans,
  //   please consider using a `bit_set` instead.
  // * Odin's `bit_field` and C's bit-fields might not be compatible
  //   * Odin's `bit_field`'s have a well defined layout (Least-Significant-Bit)
  //   * C's bit-fields on `struct`'s are undefined and are not portable across targets and compilers
  // * A `bit_field`'s field type can only be one of the following:
  //   * Integer
  //   * Boolean
  //   * Enum
}

main :: proc() {
  /*
  For More Odin Examples - https://github.com/odin-lang/examples
  This repository contains examples of how certain things can be accomplished
  in idiomatic Odin, allowing you learn its semantics, as well as how to use
  parts of the core and vendor package collections.
  */

  when true {
    the_basics()
    control_flow()
    named_proc_return_parameters()
    variadic_procedures()
    explicit_procedure_overloading()
    struct_type()
    union_type()
    using_statement()
    implicit_context_system()
    parametric_polymorphism()
    threading_example()
    array_programming()
    map_type()
    implicit_selector_expression()
    partial_switch()
    cstring_example()
    bit_set_type()
    deferred_procedure_associations()
    reflection()
    quaternions()
    unroll_for_statement()
    where_clauses()
  }
}

```

```
foreign_system()
ranged_fields_for_array_compound_literals()
deprecated_attribute()
range_statements_with_multiple_return_values()
soa_struct_layout()
constant_literal_expressions()
union_maybe()
explicit_context_definition()
or_else_operator()
or_return_operator()
or_break_and_or_continue_operators()
arbitrary_precision_mathematics()
matrix_type()
bit_field_type()
}
}
```