Variables

package main

import "fmt"

func main() {

// Declare variables that are set to their zero value.

var a int

var b string

var c float64

var d bool

fmt.Printf("var a int \t %T [%v]\n", a, a)

fmt.Printf("var b string \t %T [%v]\n", b, b)

fmt.Printf("var c float64 \t %T [%v]\n", c, c)

fmt.Printf("var d bool \t %T [%v]\n\n", d, d)

// Declare variables and initialize.

// Using the short variable declaration operator.

aa := 10

bb := "hello"

cc := 3.14159

dd := true

fmt.Printf("aa := 10 \t %T [%v]\n", aa, aa)

fmt.Printf("bb := \"hello\" \t %T [%v]\n", bb, bb)

fmt.Printf("cc := 3.14159 \t %T [%v]\n", cc, cc)

fmt.Printf("dd := true \t %T [%v]\n\n", dd, dd)

// Specify type and perform a conversion.

aaa := int32(10)

fmt.Printf("aaa := int32(10) %T [%v]\n", aaa, aaa)

}

/\*

Zero Values:

Type Initialized Value

Boolean false

Integer 0

Floating Point 0

Complex 0i

String "" (empty string)

Pointer nil

\*/

Struct Types

CR1

// Sample program to show how to declare and initialize struct types.

package main

import "fmt"

// example represents a type with different fields.

type example struct {

flag bool

counter int16

pi float32

}

func main() {

// Declare a variable of type example set to its

// zero value.

var e1 example

// Display the value.

fmt.Printf("%+v\n", e1)

// Declare a variable of type example and init using

// a struct literal.

e2 := example{

flag: true,

counter: 10,

pi: 3.141592,

}

// Display the field values.

fmt.Println("Flag", e2.flag)

fmt.Println("Counter", e2.counter)

fmt.Println("Pi", e2.pi)

}

CR2

// Sample program to show how to declare and initialize anonymous

// struct types.

package main

import "fmt"

func main() {

// Declare a variable of an anonymous type set

// to its zero value.

var e1 struct {

flag bool

counter int16

pi float32

}

// Display the value.

fmt.Printf("%+v\n", e1)

// Declare a variable of an anonymous type and init

// using a struct literal.

e2 := struct {

flag bool

counter int16

pi float32

}{

flag: true,

counter: 10,

pi: 3.141592,

}

// Display the values.

fmt.Printf("%+v\n", e2)

fmt.Println("Flag", e2.flag)

fmt.Println("Counter", e2.counter)

fmt.Println("Pi", e2.pi)

}

CR3

// Sample program to show how variables of an unnamed type can

// be assigned to variables of a named type, when they are

// identical.

package main

import "fmt"

// example represents a type with different fields.

type example struct {

flag bool

counter int16

pi float32

}

func main() {

// Declare a variable of an anonymous type and init

// using a struct literal.

e := struct {

flag bool

counter int16

pi float32

}{

flag: true,

counter: 10,

pi: 3.141592,

}

// Create a value of type example.

var ex example

// Assign the value of the unnamed struct type

// to the named struct type value.

ex = e

// Display the values.

fmt.Printf("%+v\n", ex)

fmt.Printf("%+v\n", e)

fmt.Println("Flag", e.flag)

fmt.Println("Counter", e.counter)

fmt.Println("Pi", e.pi)

}

Pointers

CR1

// Sample program to show the basic concept of pass by value.

package main

func main() {

// Declare variable of type int with a value of 10.

count := 10

// Display the "value of" and "address of" count.

println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass the "value of" the count.

increment(count)

println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

}

// increment declares count as a pointer variable whose value is

// always an address and points to values of type int.

//go:noinline

func increment(inc int) {

// Increment the "value of" inc.

inc++

println("inc:\tValue Of[", inc, "]\tAddr Of[", &inc, "]")

}

CR2

// Sample program to show the basic concept of using a pointer

// to share data.

package main

func main() {

// Declare variable of type int with a value of 10.

count := 10

// Display the "value of" and "address of" count.

println("count:\tValue Of[", count, "]\t\tAddr Of[", &count, "]")

// Pass the "address of" count.

increment(&count)

println("count:\tValue Of[", count, "]\t\tAddr Of[", &count, "]")

}

// increment declares count as a pointer variable whose value is

// always an address and points to values of type int.

//go:noinline

func increment(inc \*int) {

// Increment the "value of" count that the "pointer points to".

\*inc++

println("inc:\tValue Of[", inc, "]\tAddr Of[", &inc, "]\tValue Points To[", \*inc, "]")

}

CR3

// Sample program to show the basic concept of using a pointer

// to share data.

package main

import "fmt"

// user represents a user in the system.

type user struct {

name string

email string

logins int

}

func main() {

// Declare and initialize a variable named bill of type user.

bill := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

//\*\* We don't need to include all the fields when specifying field

// names with a struct literal.

// Pass the "address of" the bill value.

display(&bill)

// Pass the "address of" the logins field from within the bill value.

increment(&bill.logins)

// Pass the "address of" the bill value.

display(&bill)

}

// increment declares logins as a pointer variable whose value is

// always an address and points to values of type int.

func increment(logins \*int) {

\*logins++

fmt.Printf("&logins[%p] logins[%p] \*logins[%d]\n\n", &logins, logins, \*logins)

}

// display declares u as user pointer variable whose value is always an address

// and points to values of type user.

func display(u \*user) {

fmt.Printf("%p\t%+v\n", u, \*u)

fmt.Printf("Name: %q Email: %q Logins: %d\n\n", u.name, u.email, u.logins)

}

CR4

// Sample program to teach the mechanics of escape analysis.

package main

// user represents a user in the system.

type user struct {

name string

email string

}

// main is the entry point for the application.

func main() {

u1 := createUserV1()

u2 := createUserV2()

println("u1", &u1, "u2", u2)

}

// createUserV1 creates a user value and passed

// a copy back to the caller.

//go:noinline

func createUserV1() user {

u := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

println("V1", &u)

return u

}

// createUserV2 creates a user value and shares

// the value with the caller.

//go:noinline

func createUserV2() \*user {

u := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

println("V2", &u)

return &u

}

CR5

// Sample program to show how stacks grow/change.

package main

// Number of elements to grow each stack frame.

// Run with 1 and then with 1024

const size = 1

// main is the entry point for the application.

func main() {

s := "HELLO"

stackCopy(&s, 0, [size]int{})

}

// stackCopy recursively runs increasing the size

// of the stack.

//go:noinline

func stackCopy(s \*string, c int, a [size]int) {

println(c, s, \*s)

c++

if c == 10 {

return

}

stackCopy(s, c, a)

}

Constants

CR1

// Sample program to show how to declare constants and their

// implementation in Go.

package main

import "fmt"

func main() {

// Constants live within the compiler.

// They have a parallel type system.

// Compiler can perform implicit conversions of untyped constants.

// Untyped Constants.

const ui = 12345 // kind: integer

const uf = 3.141592 // kind: floating-point

// Typed Constants still use the constant type system but their precision

// is restricted.

const ti int = 12345 // type: int

const tf float64 = 3.141592 // type: float64

// ./constants.go:XX: constant 1000 overflows uint8

// const myUint8 uint8 = 1000

// Constant arithmetic supports different kinds.

// Kind Promotion is used to determine kind in these scenarios.

// Variable answer will of type float64.

var answer = 3 \* 0.333 // KindFloat(3) \* KindFloat(0.333)

fmt.Println(answer)

// Constant third will be of kind floating point.

const third = 1 / 3.0 // KindFloat(1) / KindFloat(3.0)

// Constant zero will be of kind integer.

const zero = 1 / 3 // KindInt(1) / KindInt(3)

// This is an example of constant arithmetic between typed and

// untyped constants. Must have like types to perform math.

const one int8 = 1

const two = 2 \* one // int8(2) \* int8(1)

}

CR2

// Sample program to show how constants do have a parallel type system.

package main

import "fmt"

const (

// Max integer value on 64 bit architecture.

maxInt = 9223372036854775807

// Much larger value than int64.

bigger = 9223372036854775808543522345

// Will NOT compile

// biggerInt int64 = 9223372036854775808543522345

)

func main() {

fmt.Println("Will Compile")

}

CR3

// Sample program to show how iota works.

package main

import "fmt"

func main() {

const (

A1 = iota // 0 : Start at 0

B1 = iota // 1 : Increment by 1

C1 = iota // 2 : Increment by 1

)

fmt.Println("1:", A1, B1, C1)

const (

A2 = iota // 0 : Start at 0

B2 // 1 : Increment by 1

C2 // 2 : Increment by 1

)

fmt.Println("2:", A2, B2, C2)

const (

A3 = iota + 1 // 1 : Start at 0 + 1

B3 // 2 : Increment by 1

C3 // 3 : Increment by 1

)

fmt.Println("3:", A3, B3, C3)

const (

Ldate = 1 << iota // 1 : Shift 1 to the left 0. 0000 0001

Ltime // 2 : Shift 1 to the left 1. 0000 0010

Lmicroseconds // 4 : Shift 1 to the left 2. 0000 0100

Llongfile // 8 : Shift 1 to the left 3. 0000 1000

Lshortfile // 16 : Shift 1 to the left 4. 0001 0000

LUTC // 32 : Shift 1 to the left 5. 0010 0000

)

fmt.Println("Log:", Ldate, Ltime, Lmicroseconds, Llongfile, Lshortfile, LUTC)

}

CR4

/\*

// A Duration represents the elapsed time between two instants as

// an int64 nanosecond count. The representation limits the largest

// representable duration to approximately 290 years.

type Duration int64

// Common durations. There is no definition for units of Day or larger

// to avoid confusion across daylight savings time zone transitions.

const (

Nanosecond Duration = 1

Microsecond = 1000 \* Nanosecond

Millisecond = 1000 \* Microsecond

Second = 1000 \* Millisecond

Minute = 60 \* Second

Hour = 60 \* Minute

)

// Add returns the time t+d.

func (t Time) Add(d Duration) Time

\*/

// Sample program to show how literal, constant and variables work

// within the scope of implicit conversion.

package main

import (

"fmt"

"time"

)

func main() {

// Use the time package to get the current date/time.

now := time.Now()

// Subtract 5 nanoseconds from now using a literal constant.

literal := now.Add(-5)

// Subtract 5 seconds from now using a declared constant.

const timeout = 5 \* time.Second // time.Duration(5) \* time.Duration(1000000000)

constant := now.Add(-timeout)

// Subtract 5 nanoseconds from now using a variable of type int64.

minusFive := int64(-5)

variable := now.Add(minusFive)

// example4.go:50: cannot use minusFive (type int64) as type time.Duration in argument to now.Add

// Display the values.

fmt.Printf("Now : %v\n", now)

fmt.Printf("Literal : %v\n", literal)

fmt.Printf("Constant: %v\n", constant)

fmt.Printf("Variable: %v\n", variable)

}