**Note:**

* Video Lecture M2.1.2: At 0:55, the instructor says "x=1" but should actually say "x=4." The screen text is correct.
* Video Lecture M2.2.2: Note that UTF-8 (Unicode Transformation Format - 8) is an encoding standard for Unicode along with UTF-16 and UTF-32. This latter standard is a fixed-width encoding, in which each 32-bit value represents one Unicode code point. Some programmers have UTF-32 in mind when they think about "Unicode." For more details about Unicode and UTF, please visit the FAQ page on the Unicode website: <http://unicode.org/faq/utf_bom.html> .
* Video Lecture 2.2.3: At 5:14, the text on the screen shows itoa(s) but it should be itoa(i).

**Module 1 – 1 Pointers**

1. Pointers

* A pointer is an address to data in memory.
* & operator returns the address of a variable / function.
* operator returns data at an address (dereferencing).
* & and \* can be inputted in front of the variable.
* Example:

var x int = 1

var y int // initialize to 0

var ip \* knt // ip is pointer to int

ip = &x // ip now points to x

y = \*ip // y is now 1

1. New

* Alternate way to create a variable
* New() function creates a variable and returns a pointer to the variable
* Variable is initialized to zero
* Example:

ptr := new(int)

\*ptr = 3

**Module 1 – 2 Variable Scope**

1. **Variable Scope**

* The places in code where a variable can be accessed.
* Example 1:

var x = 4

func f() {

fmt.Printf(“%d”, x)

}

func g() {

fmt. Printf (“%d”, x)

}

* Example 2:

func f() {

var x = 4

fmt. Printf (“%d”, x)

}

func g() {

fmt. Printf (“%d”, x)

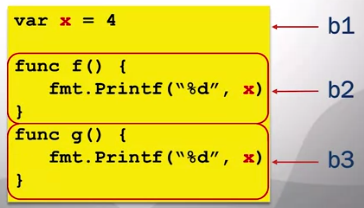
}

1. **Blocks**

* In Go, variable scoping is done using blocks.
* A sequence of declarations and statements within matching brackets {}, including function definitions
* Hierarchy of implicit blocks also
* Universe block – all Go source
* Package block – all source in a package
* Fileblock – all source in a file
* “if”, “for”, “switch” – all code inside the statement
* Clause in “switch” or “select” – individual clauses each get a block

1. **Lexical Scoping**

* Go is lexically scoped language using blocks, lexical means we talk about the relationship of one block being defined inside another block
* bi >= bj if bj id defined inside bi, “defined inside” is transitive,so if bj is defined inside bi, so bj would be bi >= bj

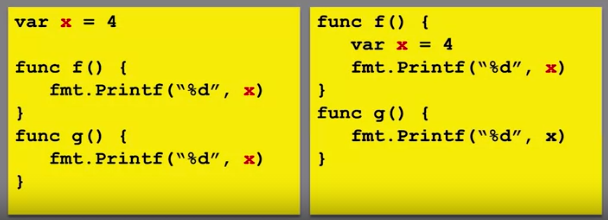


Based on the example above, b2 and b3 are defined inside b1.

1. **Scope of Variables**

* Variable accessible from block bj if

1. Variable is declared in block bi, and
2. bi >= bj



**Module 1 – 3 Deallocating Memory**

1. **Deallocating Space**

* When a variable is no longer needed, it should be deallocated. Memroy space is made available.
* Otherwise, we will eventually run out of memory

func f() {

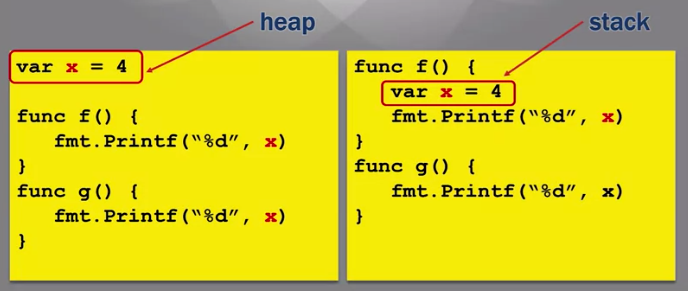
var x = 4

fmt.Printf(“%d”, x)

}

* Each call f() creates an integer
* Now, you call in your program the function f 100 times, then it is going to allocate 100 different spaces for this variable X because the X goes away after the function completes, the X goes away ant it is going to allocate it again, and so on.

1. Stack vs Heap



* Stack is dedicated to function calls
* Local variables are store here
* Deallocated after function completes
* Heap is persistent

1. Manual Deallocation (like in C language)

* Data on the heap must be deallocated when it is done being used.
* In most compiled languages (i.e. C), this is done manually.
* Example:

x= malloc(32);

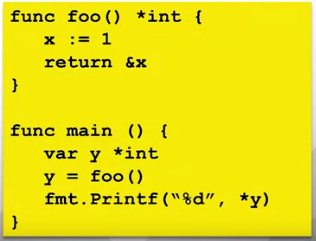
free(x);

* It will allocate 32 bytes of memory and X will be appointed to that.
* Error prone, but fast.

**Module 1 – 4 Garbage Collection**

1. **Pointers and Deallocation**

* Hard to determine when a variable is no longer in use, the problem is because we can only deallocate the variable if no longer in use, so it is hard to determine when the variable will be used or no longer in use.
* Example:



Function foo return the pointer x (not the data) into main. We can not say foo is done, because main is going to use that local variable because now main has a pointer to it. This I a legal thing to do in Go.

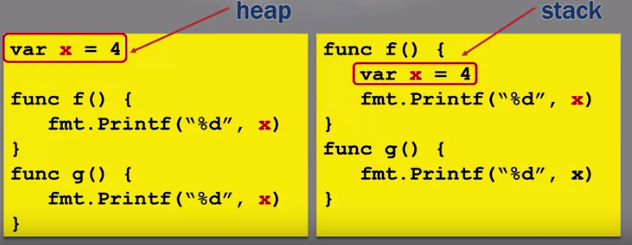
* Conclusion: foo() returns pointer to x.

1. **Garbage Collection**

* In interpreted languages, this is done by the interpreter. (To determine when a variable is not in use anymore and when it is)
  + Java Virtual Machine
  + Python Interpreter
  + Easy for the programmer
  + Slow (need an interpreter), it requires an interpreter so generally compiled languages like C, C++, they can not do it, but Go is different.

1. **Garbage Collection in Go**

* Go is a compiled language which enables garbage collection
* Implementation is fast



* Compiler determines stack vs heap. It will actually allocate stack on the heap and the stack itself.
* Garbage collection in the background.

**Module 2 – 1 Comments, Printing, Integers**

1. **Comments**

* Comments are text for understandability
* Ignored by the compiler
* Single-line comments

Examples:

// This is a comment

var x int // Another comment

* Block comments

Examples:

/\* Comment1

Comment2 \*/

var x int

1. **Printing**

* Import from the fmt package

fmt.Printf() (fmt.Println)

* Example: prints a string

fmt.Printf(“Hi”)

x := “Joe”

fmt.Printf(“Hi “ + x)

* Format strings are good for formatting
* Conversion characters for each argument

fmt.Printf(“Hi %s”, x)

1. **Integers**

* Generic int declaration

var x int

* Different lengths and signs

Int8, int16, int32, int64, uint8, uint16, uint32, uint64

* Unsigned integers most significant bit is not use for the sign, it is used for magnitude representation. The magnitude, the abolute value of unsigned integer can get bigger just because you have that extra bit that would have been used for signing and irregular integer.
* Binary operators
* Arithmetic: + - \* / % << >>
* Comparison: == != > < >= <=
* Boolean: && ||

**Module 2 – 2 Ints, Floats, Strings**

1. **Type Conversions**

* Most binary operations need operands of the same type, including assignments. The example below is fail because the integers has different types of integers.

var x int32 = 1

var y int16 = 2

x = y

* Convert type with T() operation

x = int32(y)

1. **Floating Point**

* float32 - ~6 digits of precision
* float64 - ~ 15 digits of precision
* Expressed using decimals or scientific notation

Var x float64 = 123.45

Var y float64 = 1.2345e2

* Complex numbers represented as two floats: real and imaginary

Var z complex128 = complex(2, 3) // 2 is real and 3 is imaginary

1. **ASCII and Unicode**

* American Standard Code for Information Exchange
* Character coding – each character is associated with an (7) 8-bit number

‘A’ = 0x41

* ASCII is an 8-bit long code, which means it can maximum represent 256 possible characters. We can use 8-bit code to represent Chinese language.
* **Unicode** is a 32-bit character code.
* **UTF-8** is variable length, 8-bit UTF codes are same as ASCII, it is 8 bit and it can go up to 32 bit.
* **Code points** – Unicode characters, it can be up to 2 to the 32 code points.
* **Rune** – a code point in Go.

1. **Strings**

* Sequence of arbitrary bytes
* Read only
* Often meant to be printed
* String literal – notated by double quotes
* X := “Hi there”
* Each byte is a rune (UTF-8 code point)

**Module 2 – 3 String Packages**

1. **Unicode Package**

* Runes are divided into many different categories
* Provides a set of functions to test categories of runes
* IsDigit(r rune)
* IsSpace(r rune)
* IsLetter(r rune)
* IsLower(r rune)
* IsPunct(r rune)
* Some functions perform conversions
* ToUpper(r rune)
* ToLower(r rune)

1. **Strings Package**

* Functions to manipulate UTF-8 encoded strings.
* String search functions
* Compare(a, b) – returns an integer comparing two strings.
* Lexicographically. 0 if a == b, -1 if a < b, and +1 if a > b.
* Contains(s, substr) – returns true if substring is inside s.
* HasPrefix(s, prefix) – returns true if the string s beings with prefix.
* Index(s, substr) – returns the index of the first instance of substr in s

1. **String Manipulation**

* Strings are immutable, but modified strings are returned.
* Replace(s, old, new, n) – replace returns a copy of the string s with the first n instances of old replaced by new.
* ToLower(s)
* ToUpper(s)
* TrimSpace(s) – returns a new string with all leading and trailing white space removed.

1. **Strconv Package**

* Conversions to and from string representations of basic data types.
* Atoi(s) – converts string s to int
* Itoa(s) – converts int (base 10) to string
* FormatFloat (f, fmt, prec, bitSize) – convers floating point number to a string
* ParseFloat (s, bitSize) – converts a string to a floating point number

**Module 3 – 1 Constants**

1. **Constants**

* Expressions whose value is known at compile time
* Type is inferred from righthand side (boolean, string, number)
* Example:

const x = 1.3

const (

y = 4

z = “Hi”

)

1. iota

* Generate a set of related but distinct constants
* Often represents a property which has several distinct possible values. Example: days of the week, months of the year.
* Constants must be different but actual value is not important. We don’t care if Monday is 500, and Tuesday is 5000, we just care about the value of Monday and Tuesday must be different.
* Like an enumerated type in other languages.
* Example:

type Grades int

const (

A Grades = iota

B

C

D

F

)

* Each constant is assigned to a unique integer
* Starts at 1 and increments

**Module 3 – 2 Control Flow**

1. **Control Structures**

* Statements which alter control flow

If <condition> {

<consequent>

}

* Expression <condition> is evaluated
* <consequent> statements are executed if condition is true

If x > 5 {

fmt.Printf(“Yup”)

}

1. **For Loops**

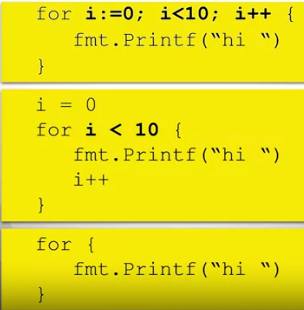
* Iterates while a condition is true
* May have an initialization and update operation

for <init>; <cond>; <update> {

<stmts>

}

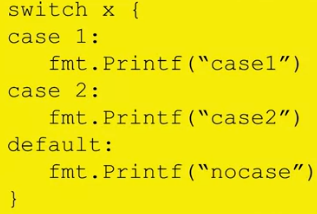
* Example:



The last is an infinite loop, because we don’t have the condition and update value.

1. **Switch / Case**

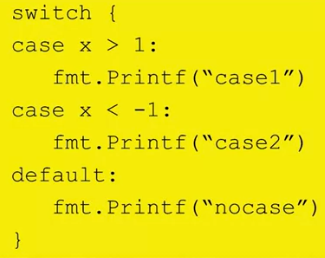
* Switch is a multi way if statement
* Swithc may containa tag which is a variable to be checked
* Tag is compared to a constant defined in each case
* Case which matches tag is executed
* Example:



**Module 3 – 3 Control Flow, Scan**

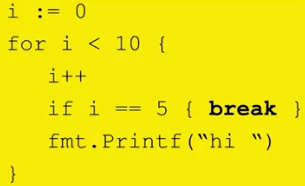
1. **Tagless Switch**

* Switch may not contain a tag
* Case contains a booelan expression to evaluate
* First true case is executed
* Example:

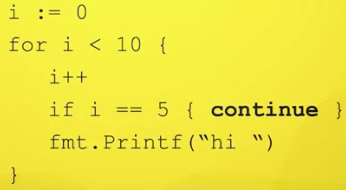


1. **Break and continue**

* Break exits the containing loop
* Example:



* Continue skips the rest of the current iteration
* Example:



1. **Scan**

* Scan reads user input
* Takes a pointer as an argument
* Typed data is written to pointer
* Returns number of scanned items
* Example:

var appleNum int

fmt.Printf(“Number of apples?”)

num, err :=

fmt.Scan(&appleNum)

fmt.Printf(appleNum)

**Module 3 – 3 Assignment 2**

**Answer 1 Submit**

package main

import (

    "fmt"

)

func main() {

    var input\_number float64

    fmt.Printf("Input your number: ")

    fmt.Scan(&input\_number)

    fmt.Println(int(input\_number))

}

**Answer 2 Complete Answer**

package main

import (

    "fmt"; "strconv"

)

func main() {

    var input\_number float64

    fmt.Printf("Input your number: ")

    num, \_ :=   fmt.Scan(&input\_number)     // num is the pointer

    s := fmt.Sprintf("%f", input\_number)    // Convert float64 to String

    fmt.Println(s)                          // Print String value

    fmt.Println(int(input\_number))          // Convert float64 to Integer, Print

    fmt.Println(strconv.Itoa(num))          // Convert Integer to String

}