###: Course Title

Lecture title

Lecturer name
The Go programming language

Go is an imperative, statically typed, natively compiled and concurrent programming language.

See more at golang.org
Go source code are organised in packages (e.g. main)
The Go programming language

Basic program structure

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

**Import path** outside standard library are identified uniquely e.g. "github.com/pkg/errors"
The Go programming language

Basic program structure

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!"
}
```

The main() function in the main package is the program entry point.
The Go programming language

Basic program structure

```go
default package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Access exported identifiers from an imported package using its package name, e.g. errors.New
The Go programming language
Building and running

Build a Go program

```
$ go build hello.go  # creates 'hello'
$ ./hello           # run 'hello' binary
Hello, world!
$
```

Alternatively, build and run directly with go run

```
$ go run hello.go
Hello, world!
$
```
func main() {
    sum := 0

    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println("Sum of 0..9", sum)

    if isEven(sum) {
        fmt.Println("Sum is even")
    } else {
        fmt.Println("Sum is odd")
    }
}

func isEven(v int) bool {
    return v%2 == 0
}
The Go programming language

Control flow

```go
func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println("Sum of 0..9", sum)
    if isEven(sum) {
        fmt.Println("Sum is even")
    } else {
        fmt.Println("Sum is odd")
    }
}

func isEven(v int) bool {
    return v%2 == 0
}
```

- C-style `for-loop` without parenthesis
- Short variable declaration, type automatically inferred, equivalent to `var sum int = 0`
- `for`-loop without parenthesis around condition
- Function definition with `int` parameter and returns a `bool`
func main() {
    sum := 0

    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println("Sum of 0..9", sum)

    if isEven(sum) {
        fmt.Println("Sum is even")
    } else {
        fmt.Println("Sum is odd")
    }
}

func isEven(v int) bool {
    return v%2 == 0
}
func main() {
    sum := 0

    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println("Sum of 0..9", sum)

    if isEven(sum) {
        fmt.Println("Sum is even")
    } else {
        fmt.Println("Sum is odd")
    }
}

func isEven(v int) bool {
    return v%2 == 0
}
The Go programming language

Types

**Primitive types**
- Boolean, numeric, string, `rune`, byte

**Composite types**
- Data structures: e.g. array, slice, struct, maps
- Channel types
- Function and interface types
The Go programming language

Types

### Primitive types

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Integer</td>
<td>int</td>
</tr>
<tr>
<td>Floating point</td>
<td>float32</td>
</tr>
<tr>
<td>Unicode code point</td>
<td>rune</td>
</tr>
<tr>
<td>Byte (8-bit)</td>
<td>byte</td>
</tr>
</tbody>
</table>

Example

```go
var i int // Declare i as int variable
i := 42    // Or by short var declaration (type inferred)
```
The Go programming language

Types

Data structures

- **Struct**: Collection of named fields
- **Array**: Fixed size numbered sequence
- **Slice**: Variable size numbered sequence
- **Map**: Unordered associative array
The Go programming language

Types

Data structures: Struct

Example

```go
// A type declaration for a struct
type book struct {
    title string // A field per line
    authors string
    year int
}

func main() {
    goBook := book{
        title: "The Go Programming Language",
        authors: "Alan Donovan and Brian Kernighan",
        year: 2015,
    }
    fmt.Println(goBook.title, "is written by", goBook.authors)
}
```

Initialise a value of struct type

Access a struct field with .fieldName
Data structures: *Array*

- Fixed size numbered sequence
- Index is non-negative integer (starts from 0)
- Access elements using the index

Example

```go
var x [3] int
x[0] = 7
x[1] = 2
x[2] = 3

xlen := len(x)

fmt.Println("x[1] is", x[1]) // 2
fmt.Println("length is ", xlen) // 3
```

Built-in function `len(s)` gives the length of the array (or slice) `s`; The last valid index is `len(s)-1`
Data structures: Array

Example

```go
var x [3]int
x[0] = 7
x[1] = 2
x[2] = 3

xlen := len(x)

fmt.Println("x[1] is", x[1]) // 2
fmt.Println("length is ", xlen) // 3
```

Built-in function `len(s)` gives the length of the array (or slice) `s`; The last valid index is `len(s)-1`
Data structures: Slice

- Initialise with `make([]T, length)`
- Usage similar to array
- Extend with `append(slice, elem)`

Example

```go
y := make([]int, 4) // y[0] = ...
ylen := len(y)
y = append(y, 42)

z := y[1:5]

fmt.Println("old len(y) is", ylen) // 4
fmt.Println("new len(y) is", len(y)) // 5
fmt.Println("len(z) is", len(z)) // 4
```

Returns initialised `int` slice with length 4, slice empty by default
Data structures: Slice
- Initialise with `make([]T, length)`
- Usage similar to array
- Extend with `append(slice, elem)`

Example

```go
y := make([]int, 4) // y[0] = ...
ylen := len(y)
y = append(y, 42)

z := y[1:5]

fmt.Println("old len(y) is", ylen)  // 4
fmt.Println("new len(y) is", len(y)) // 5
fmt.Println("len(z) is", len(z))    // 4
```

Returns slice with 42 appended to the end and length extended
The Go programming language

Types

Data structures: Slice
- Initialise with \texttt{make([]T, length)}
- Usage similar to array
- Extend with \texttt{append(slice, elem)}

Example

\begin{verbatim}
y := \texttt{make}([]\texttt{int}, 4) // y[0] = ...
ylen := \texttt{len}(y)
y = \texttt{append}(y, 42)

z := y[1:5]

fmt.Println("old \texttt{len(y) is}", ylen) // 4
fmt.Println("new \texttt{len(y) is}", \texttt{len}(y)) // 5
fmt.Println("\texttt{len(z) is}", \texttt{len}(z)) // 4
\end{verbatim}
Data structures: Slice

Example

```go
y := make([]int, 4)
y := append(y, 42)
z := y[1:5]
```
The Go programming language

Types

Data structures: Slice

Example

```go
y := make([]int, 4)

len 4
cap 4

y[0] [1] ... [len(y)-1]

int int int int

y' := append(y, 42)

len 5
cap 5

y'[0] [1] ... [len(y')-1]

int int int int int 42

z := y'[1:5]

len 4
cap 4

z[0] [1] ... [len(z)-1]

int int int 42
```
The Go programming language

Types

Data structures: Map
- Unordered key-value pairs
- Index by key type with unique values

Example

```go
m := make(map[int]string) // Returns initialised map with key type int & element type string
m[1] = "value"
strVal = m[1]

// Test for key
value, exists := m[1]
if exists {
    // 1 is in the key
    fmt.Println("value found:", value)
}

delete(m, 1)
```
The Go programming language

Types

Data structures: Map

- Unordered key-value pairs
- Index by key type with unique values

Example

```go
m := make(map[int]string) // Insert value into the map
m[1] = "value"
strVal = m[1]

// Test for key
value, exists := m[1]
if exists {
    // 1 is in the key
    fmt.Println("value found:", value)
}

delete(m, 1)
```
The Go programming language

Types

Data structures: Map

- Unordered key-value pairs
- Index by key type with unique values

Example

m := make(map[int]string)

m[1] = "value"
strVal = m[1]

// Test for key
value, exists := m[1]
if exists {
    // 1 is in the key
    fmt.Println("value found:", value)
}

delete(m, 1)
Data structures: Map

- Unordered key-value pairs
- Index by key type with unique values

Example

```go
m := make(map[int]string)

m[1] = "value"
strVal = m[1]

// Test for key
value, exists := m[1]
if exists {
    // 1 is in the key
    fmt.Println("value found:", value)
}
```

Lookup returns optional boolean indicating if lookup was successful.
Data structures: Map

- Unordered key-value pairs
- Index by key type with unique values

Example

```go
m := make(map[int]string)
m[1] = "value"
strVal = m[1]

// Test for key
value, exists := m[1]
if exists {
    // 1 is in the key
    fmt.Println("value found:", value)
}

// Delete value at given key
delete(m, 1)
```
The Go programming language

Types

Channel type

// make(chan T) returns
// an initialised chan T
ch := make(chan int)

// Send through a channel
value := 42
ch <- value

// Receive from a channel
received := <-ch

Channel creation

Uninitialised int channel can be declared as var ch chan int
Channel type

// make(chan T) returns
// an initialised chan T
ch := make(chan int)

// Send through a channel
value := 42
ch < value

// Receive from a channel
received := ch

Send blocks if channel is full. <- points from value to channel

Receive blocks if channel is empty. <- points from the channel, storing received value not mandatory
The Go programming language

Concurrency

Goroutines
- Lightweight *threads*
- Functions, executed concurrently

Channels
- *Typed* FIFO queues for message passing
  - e.g. `chan int` can send/receive `int`
- Can be accessed concurrently by goroutines
Concurrent in Go
Simple message passing

```go
func main() {
    ch := make(chan int)
    chEnd := make(chan int)

    go sender(42, ch)
    go receiver(ch, chEnd)
    <-chEnd
    fmt.Println("All done.")
}

func sender(v int, ch chan int) {
    ch <- v
}

func receiver(ch, rcvd chan int) {
    value := <-ch
    rcvd <- value // Notifies receive complete
}
```

A **channel** for sender and receiver to communicate
Concurrent in Go

Simple message passing

```go
func main() {
    ch := make(chan int)
    chEnd := make(chan int)

    go sender(42, ch)
    go receiver(ch, chEnd)
    <-chEnd
    fmt.Println("All done.")
}

func sender(v int, ch chan int) {
    ch <- v
}

func receiver(ch, rcvd chan int) {
    value := <-ch
    rcvd <- value // Notifies receive complete
}
```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    chEnd := make(chan int)

    go sender(41, ch1)
    go sender(42, ch2)
    go receiver(ch1, ch2, chEnd)
    <-chEnd
    fmt.Println("All done.")
}

func receiver(ch1, ch2, rcvd chan int) {
    var value int
    select {
    case value = <-ch1: // value is 41
    case value = <-ch2: // value is 42
    }
    close(rcvd)
}
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    chEnd := make(chan int)

    go sender(41, ch1)
    go sender(42, ch2)
    go receiver(ch1, ch2, chEnd)

    <-chEnd
    fmt.Println("All done.")
}

func receiver(ch1, ch2, rcvd chan int) {
    var value int
    select {
        case value = <-ch1:
            // value is 41
        case value = <-ch2:
            // value is 42
    }
    close(rcvd)
}
Further topics

- OOP with method receivers
- Interface types
- for-range statements
Issues in concurrent systems

Concurrent systems are complex
- Each component executes concurrently
- Coordination by message passing
Concurrency could lead to different class of issues

Safety & liveness properties in concurrent systems

Properties true for every possible execution
- Safety: nothing bad happens
- Liveness: something good eventually happens
To understand concurrent issues
Try modelling a real-world concurrent systems in Go:

Example
Consider a restaurant where a customer can order food and have food delivered.

- Customer and kitchen are concurrent
- Customer sends *order* to kitchen
- Kitchen replies with *food* to customer
func main() {
    orderCh := make(chan string)
    foodCh := make(chan food)
    go Kitchen(orderCh, foodCh)

    orderCh <- "Burger"
    burger := <- foodCh
    chips := <- foodCh

    fmt.Println("I got my", burger, "and", chips, "!")
}

func Kitchen(orderCh chan string, foodCh chan food) {
    // receives order
    order := <-orderCh
    // prepares and sends food
    preparedFood := food{name: order}
    foodCh <- preparedFood
}

Issues in concurrent systems
What is wrong with this program?
func main() {
    orderCh := make(chan string)
    foodCh := make(chan food)
    go Kitchen(orderCh, foodCh)

    orderCh <- "Burger"
    burger := <-foodCh
    chips := <-foodCh

    fmt.Println("I got my", burger, "and", chips, "!")
}

Main (customer) goroutine

- Sends one order (to orderCh)
- Expects two food delivery (from foodCh)
Issues in concurrent systems

What is wrong with this program?

Restaurant goroutine

- Accepts orders (from orderCh)
- Delivers food (to foodCh)

```go
func Kitchen(orderCh chan string, foodCh chan food) {
    // receives order
    order := <-orderCh
    // prepares and sends food
    preparedFood := food{name: order}
    foodCh <- preparedFood
}
```

Receive order
Send food
Deadlock

All goroutines in the program are either *completed* or *blocked waiting for a channel* (stuck)
Issues in concurrent systems

Attempt 2: Fix the deadlocked program

```go
func main() {
    orderCh := make(chan string)
    foodCh := make(chan food)
    go Kitchen(orderCh, foodCh)

    orderCh <- "Burger"
    burger := <-foodCh

    fmt.Println("I got my", burger, "!")
}
```

Only receive food **once** for one order
However, now the kitchen becomes very busy

```go
func Kitchen(orderCh chan string, foodCh chan food) {
    kitchenIsBusy := true // kitchen is always busy

    // receives order
    order := <-orderCh

    for kitchenIsBusy {
        // do other work
    }

    // prepares and sends food
    preparedFood := food{name: order}
    foodCh <- preparedFood
}
```

Kitchen never becomes available

So, will the customer *eventually* receive their food?
Issues in concurrent systems

However, now the kitchen becomes very busy

```
func Kitchen(orderCh chan string, foodCh chan food) {
    kitchenIsBusy := true // kitchen is always busy

    // receives order
    order := <-orderCh

    for kitchenIsBusy {
        // do other work
    }

    // prepares and sends food
    preparedFood := food{name: order}
    foodCh <- preparedFood
}
```

So, will the customer eventually receive their food? (No)
Issues in concurrent systems

Liveness property

Something good *eventually* happens
Issues in concurrent systems

What is the value of $x$?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Shorthand for $val = val + 1$
Shorthand for $val = val - 1$
Issues in concurrent systems

What is the value of $x$?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
<th>value of $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>1</td>
</tr>
</tbody>
</table>

$x$ is 1.
Issues in concurrent systems

What is the value of $x$?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule

<table>
<thead>
<tr>
<th>Value of $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
</tr>
</tbody>
</table>

Value read via pointer
Issues in concurrent systems

What is the value of \( x \)?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule value of \( x \)

- Sequential

Value written via pointer
func main() {
  x := 1
  go inc(&x)
  go dec(&x)

  fmt.Println(x)
}

func inc(x *int) {
  val := *x
  val++
  *x = val
}

func dec(x *int) {
  val := *x
  val--
  *x = val
}
Issues in concurrent systems

What is the value of \( x \)?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}
```

```go
func inc(x *int) {
    val := *x
    val++
    *x = val
}
```

```go
func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Value of ( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>( x = 1 )</td>
</tr>
</tbody>
</table>

Value written via pointer:

- \( *x = 1 \) (val = 2)
- \( *x = 2 \) (val = 1)
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
Issues in concurrent systems

What is the value of $x$?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)

    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

<table>
<thead>
<tr>
<th>Schedule</th>
<th>value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>1</td>
</tr>
</tbody>
</table>
Issues in concurrent systems

What is the value of $x$?

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)
    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule | value of $x$
---|---
sequential | 1

value read via pointer

value of $x$
Issues in concurrent systems

What is the value of $x$?

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Value of $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>1</td>
</tr>
<tr>
<td>interleaved</td>
<td>0 or 2</td>
</tr>
</tbody>
</table>

```go
func main() {
    x := 1
    go inc(&x)
    go dec(&x)

    fmt.Println(x)
}

func inc(x *int) {
    val := *x
    val++
    *x = val
}

func dec(x *int) {
    val := *x
    val--
    *x = val
}
```

Schedule:

- Sequential: Value of $x$ is 1
- Interleaved: Value of $x$ is 0 or 2
Data races

Concurrent (nondeterministic) access of shared resources may lead to *inconsistent program state*

- Use locks to provide atomic access
- Use message passing to control access (scalable)
Issues in concurrent systems

Models

Issues like **deadlock** or violation of **liveness properties**
- Often hidden deep inside concurrent system
- Need *abstractions* to simplify representation
- Abstractions help reveal true causes

One such abstract model is **process calculi**
e.g. Milner’s Calculus of Communicating Systems (CCS)
Issues in concurrent systems

Models

To be continued.