```
15-440, Fall 2011, Class 04, Sept. 6, 2012
Randal E. Bryant

All code available in:
    /afs/cs.cmu.edu/academic/class/15440-f12/code/class04

Go programming

Useful references:
       http://golang.org/doc/
       Ivo Balbaert, "The Way to Go", iUniverse, 2012
       Mark Summerfield, "Programming in Go", Addison-Wesley, 2012

Background:
        Robert Griesemer (don’t know him)
        Rob Pike, Ken Thompson.  Unix pioneers @ Bell Labs.  Now at Google.

Philosophical (both stated and unstated)

* Strongly typed
  -- Avoids risks of C
  -- Lets compiler detect many program errors

* Dynamic allocation with garbage collection
  -- Avoids pitfalls of managing allocation

* Avoid redundant work
  -- Doesn’t require separate interface declarations
     * Compiler extracts interface directly from code
     * Distinction of global vs. local determined by first character of name
  -- Variable declarations (rely on type inference)

Example code:

type MyStruct struct {          # Type Declaration
     iField int                 # Note reversed ordering & lack of semicolons
     Rfield float32             # Case of field selectors matters
}

[Contrast to C declaration:

typedef struct {
        int iField;
        float Rfield;
} MyStruct

# Go:
# No semicolons
# Ordering of type vs. name reversed
# Upper vs. lower case names matters
]

ms := MyStruct{1, 3.14}         # Automatically determines that ms of type MyStruct

// Pointer to structure
p := &MyStruct{Rfield:15.0}     # Like C, & takes address of something.  Can use it any
where
                                # Field Rfield set to 15.0, iField set to 0

// Field selection same either way
p.iField = ms.iField            # Note mixing of pointer vs structure.  Compiler figure
s this out


// Alternative
```

```
var q *MyStruct = new(MyStruct) # New does allocation & returns pointer.  Like malloc
q.Rfield = 15.0


* Handy features
  -- Minimize distinction between pointer and object pointed to
  -- Most semicolons inferred automatically
  -- Multi-assignment, multiple return values
  -- Order of type declarations reversed
  -- Simpler and more powerful loop & switch statements.
  -- Blank identifier

* Avoid limitations / weaknesses / risks of C(++)
  -- Lack of bounds checking on arrays
  -- Mutable strings
  -- Ability / need to do casting
  -- Nuances of signed vs. unsigned & other arithmetic type issues
  -- Separate boolean type.


* Powerful built-in data types
  -- Variable length arrays (slices)
  -- Dictionaries (maps)

* Cleaner concurrency
  -- Designed from outset to support multicore programming
  -- Low multithreading overhead
     + But carries limitation of non-preemptive scheduling

* Benefits of OO, while avoiding arcane & inefficient features
  -- Objects, but no type hierarchy
  -- "Generics" using dynamic type checking

* Important capabilities
  -- Slices: Variable length sequences
  -- Maps: Dictionaries
  -- Generic interfaces, rather than class hierarchy
  -- Control via switch & for + range
```

Let’s look at some code examples:

bufb: Implementation of FIFO buffer using linked list + tail pointer.  Single-threaded operation

Operations:

NewBuf: Create new buffer
Insert: Insert element into buffer
Front: Get first element in buffer without changing buffer
Remove: Get & remove first element from buffer
Flush: Clear buffer

```
// Linked list element
type BufEle struct {                    # Structure definition
        val []byte                      # []byte is a slice of bytes
        next *BufEle
}

func NewBuf() *Buf {                    # Returns buffer with both pointers = nil
        return new(Buf)
}

func (bp *Buf) Insert(val []byte) {     # Declaration gives something like methods
        ele := &BufEle{val : val}       # Note allocation plus taking address.  This is
 OK in Go
        if bp.head == nil {             # Standard implementation of list with tail poi
nter
                // Inserting into empty list
                bp.head = ele
                bp.tail = ele
        } else {
                bp.tail.next = ele
                bp.tail = ele
        }
}


func (bp *Buf) Remove() ([]byte, error) {
        e := bp.head
        if e == nil {
                err := errors.New("Empty Buffer")
                return nil, err
        }
        bp.head = e.next
        // List becoming empty
        if e == bp.tail { bp.tail = nil }
        return e.val, nil
}
```

Shows typical style for reporting errors: functions have return value
of type err.  When non-nil, this indicates that something went wrong.
String representing error message included.

Rest of code straightforward

```
Writing test code.

Write code in file "bufb_test.go" in same directory
Include test function(s) named TestXXXX
Run go test

// Convert integer to byte array            # Demonstration of JSON marshaling.  Trivi
al case
func i2b(i int) []byte {
        b, _ := json.Marshal(i)             # Note multi assignment and blank identifi
er
        return b
}

// Convert byte array back to integer
func b2i(b []byte) int {
        var i int
        json.Unmarshal(b, &i)
        return i
}

func TestBuf(t *testing.T) {                        # Called by test code.  Must have singl
e argument
        // Run same test ntest times
        for i := 0; i < ntest; i++ {            # Note for loop.  Like C, but no parent
heses
                bp := NewBuf()
                runtest(t, bp)
                if !bp.Empty() {
                        t.Logf("Expected empty buffer")
                        t.Fail()
                }
        }
}

func runtest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
        for removed < nele {                    # Note for loop is like while loop
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)   # Cannot insert if have done all insert
ions
                if inserted > removed && rand.Int31n(2) == 0 {
                                insert = false  # Randomly choose whether to insert or
remove
                }
                if insert {
                        bp.Insert(i2b(inserted))
                        inserted ++
                } else {
                        b, err := bp.Remove()
                        if err != nil {
                                t.Logf("Attempt to remove from empty buffer\n")
                                t.Fail()
                        }
                        v := b2i(b)
                        if v != removed {
                                t.Logf("Removed %d.  Expected %d\n", v, removed)
                                t.Fail()
                        }
                        removed ++
                }
        }
}
```

Weakness of this code: Requires data in byte slices.  Can always use marshaling, but th
at
seems inelegant.

```
Using interface types.  Go’s version of templates / generics
File bufi.go
Same idea, but use dynamically-typed buffer data

Implementation: Interface data dynamically typed.  Carries type information with it.

Operation x.(T) converts x to type T if possible, and fails otherwise.


// Linked list element
type BufEle struct {
        val interface{}         # interface defines required capabilities of val.  None
 here
        next *BufEle
}

Rest of code basically the same

Now look at testing

Case 1: Feed slices of byte arrays.
func btest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
        fmt.Printf("Byte array data: ")
        for removed < nele {
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)
                if inserted > removed && rand.Int31n(2) == 0 {
                                insert = false
                }
                if insert {
                        bp.Insert(i2b(inserted))        # Nothing special required here
                        inserted ++
                } else {                                # This is interesting
                        x, err := bp.Remove() // Type = interface{}
                        if err != nil {
                                t.Logf("Attempt to remove from empty buffer\n")
                                t.Fail()
                        }
                        # Type assertion: x is not nil & is of designated type
                        b := x.([]byte)   // Type = []byte        # Assign type to valu
e.
                        # Can also use form b, ok := x.([]byte)
                        v := b2i(b)
                        if v != removed {
                                t.Logf("Removed %d.  Expected %d\n", v, removed)
                                t.Fail()
                        }
                        removed ++
                }
        }
        fmt.Printf("Empty buffer %d/%d times\n", emptycount, nele)
}

Same thing, but for integer data.  Just look at conversion part

                        x, err := bp.Remove()  // Type = interface{}
                        ...
                        v := x.(int)      // Type = int
```

More interesting: Use random choices on type.  Code must figure out type of object:

```
                        # Insertion
                        if rand.Int31n(2) == 0 {
                                // Insert as integer
                                bp.Insert(inserted)
                        } else {
                                // Insert as byte array
                                bp.Insert(i2b(inserted))
                        }


                        # Removal
                        x, err := bp.Remove()  // Type = interface{}
                        ...
                        var iv int
                        switch v := x.(type) {
                        case int:
                                iv = v
                        case []byte:
                                iv = b2i(v)
                        default:
                                t.Logf("Invalid data\n")
                                t.Fail()
                        }
```

```
Another example: UDP proxy.  Serves as interface between server & set of clients.

For each client, maintain "connection" identifying client and connection to server.
Must come from proxy over separate port, so that server can distinguish different clien
ts

// Information maintained for each client/server connection
type Connection struct {
        ClientAddr *net.UDPAddr // Address of the client        # Note use of package "
net"
        ServerConn *net.UDPConn // UDP connection to server
}

Simple concurrency: Have different "goroutine" for each connection, to manage flow from
 server
to client

Basic scheme:
* Incoming packet from client:
  See if already have connection (look up host:port in dictionary)
      No: Create one
  Send to server along connection
* Packet from server:
  Read directly by goroutine for connection.  Sent over shared port back to client
```

```
// Global state
// Connection used by clients as the proxy server
var ProxyConn *net.UDPConn

// Address of server
var ServerAddr *net.UDPAddr

# Go map is like a dictionary.  Mapping from one type to another.
# Can map most "flat" types.  Not structures.  So, convert client host + port into stri
ng

# Reference structures allocated via "make" (not "new")
// Mapping from client addresses (as host:port) to connection
var ClientDict map [string] *Connection = make(map[string] *Connection)

# Need to protect dictionary with lock, since will have concurrent access
# We'll see in future lesson how to use Go-style concurrency.  For now,
# do something like pthread mutex.

// Mutex used to serialize access to the dictionary
var dmutex *sync.Mutex = new(sync.Mutex)

func dlock() {
        dmutex.Lock()
}

func dunlock() {
        dmutex.Unlock()
}


func setup(hostport string, port int) bool {
        // Set up Proxy
        saddr, err := net.ResolveUDPAddr("udp", fmt.Sprintf(":%d", port))
        if checkreport(1, err) { return false }
        pudp, err := net.ListenUDP("udp", saddr)        # Set up listening port
        if checkreport(1, err) { return false }
        ProxyConn = pudp
        Vlogf(2, "Proxy serving on port %d\n",port)     # My technique for printing sta
tus.

        // Get server address
        srvaddr, err := net.ResolveUDPAddr("udp", hostport)
        if checkreport(1, err) { return false }
        ServerAddr = srvaddr
        Vlogf(2, "Connected to server at %s\n", hostport)
        return true
}
```

Creating connection:

```
// Generate a new connection by opening a UDP connection to the server
func NewConnection(srvAddr, cliAddr *net.UDPAddr) *Connection {
        conn := new(Connection)
        conn.ClientAddr = cliAddr
        srvudp, err := net.DialUDP("udp", nil, srvAddr) # Note use of :=
        if checkreport(1, err) { return nil }           # Check error code
        conn.ServerConn = srvudp
        return conn
}


// Go routine which manages connection from server to single client
func RunConnection(conn *Connection) {
        var buffer [1500]byte           # Limit payload to 1500 bytes
        for {
                // Read from server
                n, err := conn.ServerConn.Read(buffer[0:])  # Pass slice of array
                if checkreport(1, err) { continue }
                // Relay it to client
                # WriteToUDP includes destination address as argument
                                        # Note [0:n] is very important
                _, err = ProxyConn.WriteToUDP(buffer[0:n], conn.ClientAddr)
                if checkreport(1, err) { continue }
                Vlogf(3, "Relayed '%s' from server to %s.\n",
                        string(buffer[0:n]), conn.ClientAddr.String())
        }
}
```

```
# Key routine

// Routine to handle inputs to Proxy port
func RunProxy() {
        var buffer[1500]byte
        for {
                n, cliaddr, err := ProxyConn.ReadFromUDP(buffer[0:])  # ReadFrom return
s address
                if checkreport(1, err) { continue }
                Vlogf(3, "Read ’%s’ from client %s\n",
                        string(buffer[0:n]), cliaddr.String())
                saddr := cliaddr.String()                            # Convert address
 to string
                dlock()
                conn, found := ClientDict[saddr]                     # Access dictiona
ry
                if !found {
                        conn = NewConnection(ServerAddr, cliaddr)
                        if conn == nil {
                                dunlock()
                                continue                             # Failure
                        }
                        ClientDict[saddr] = conn                     # Add entry to di
ctionary
                        dunlock()
                        Vlogf(2, "Created new connection for client %s\n", saddr)
                        // Fire up routine to manage new connection
                        go RunConnection(conn)                       # Start goroutine
                } else {
                        Vlogf(5, "Found connection for client %s\n", saddr)
                        dunlock()
                }
                // Relay to server
                ## Note use of Write instead of WriteToUDP.  Address part of connection
                _, err = conn.ServerConn.Write(buffer[0:n])
                if checkreport(1, err) { continue }
        }
}
```