中国科学院大学
University of Chinese Academy of Sciences

CS101

Digital Symbol Manipulation

Programs as Symbols

Programs to manipulate symbols

zxu@ict.ac.cn
zhangjialin@ict.ac.cn

Outline

- What are digital symbol manipulations?
- Data are digital symbols
- Programs are digital symbols
  - Write and execute your first Go programs
    - Introducing package and function
  - Display and input symbols
  - Convert strings to numbers
    - Introducing variable, array, string, loop
  - Use verb %c to implement verb %d
  - Good programming practices
  - A Fibonacci program using slice
- Computers are platforms of digital symbol manipulations

These slides acknowledge sources for additional data not cited in the textbook

3.1 null.go, hello.go

- null.go
  - Has two statements
  - Any standalone program belongs to the main package
  - The main function does nothing
- hello.go
  - fmt is a system-provided package, called library
    - Other people wrote fmt, which contains function fmt.Println
    - This program can reuse
  - Difference from math
    - A function may have side effect
      - to print out hello!

```
package main   // belongs to the main package
func main() {     // a main function that does nothing
}
```

命令提示符 command prompt

```
> go build null.go
> ./null
>
```

光标 cursor

```
package main   // belongs to the main package
import "fmt"       // import a built-in package
func main() {                 // has a main function
   fmt.Println(“hello!”)    // that prints out hello!
}
```

```
> go build hello.go
> ./hello
hello!
>
```

Compile and execute in two commands

```
> go run hello.go
hello!
>
```

Compile and execute in one command

Three functions in fmt

```
package fmt                 // It belongs to the fmt package
……
func Println(…) …{          // It contains a function Println which other
      …                     // programs can call by using fmt.Println
}
……
func Printf(…) …{           // It contains a function Printf which other
  …                         // programs can call by using fmt.Printf
}
……
func Scanf(…) …{            // It contains a function Scanf which other
     …                      // programs can call by using fmt.Scanf
}
```

The two statements do the same thing.

```
fmt.Printf("hello! %d\n",63)
fmt.Println("hello!",63)
```

They print the following output:

```
> hello! 63
```

Three functions in fmt

```
package fmt                 // It belongs to the fmt package
……
func Println(…) …{          // It contains a function Println which other
      …                     // programs can call by using fmt.Println
}
……
func Printf(…) …{           // It contains a function Printf which other
  …                         // programs can call by using fmt.Printf
}
……
func Scanf(…) …{            // It contains a function Scanf which other
     …                      // programs can call by using fmt.Scanf
}
```

The following code receives a user-entered integer in variable A.

```
var A int
fmt.Scanf("%d",&A)          // &A indicates the address of A
```

3.2 Understanding formatting verbs

- Example: symbols.go
- 5 representations with the formatting verb mechanism in Printf
  - Decimal
  - Hex
  - Binary
  - Character
  - String

Verb	Meaning	Example
%b	Binary	fmt.Printf("%b",63) outputs 111111
%c	Character	fmt.Printf("%c",63) outputs ?
%d	Decimal	fmt.Printf("%d",63) outputs 63
%s	String	fmt.Printf("%s",string(63)) outputs ?
%x or %X	Hexadecimal	fmt.Printf("%X",63) outputs 3F

```
package main   // symbols.go
import "fmt"
func main() {
    fmt.Printf("Decimal: %d\n",63)
    fmt.Printf("Hex: %X\n",63)
    fmt.Printf(“Binary: %b\n",63)
    fmt.Printf("Character: %c\n",63)
    fmt.Printf("String: %c%c\n",63)
    fmt.Printf("String: %c%c\n",6,3)
    fmt.Printf("String: %c%c\n",6+'0',3+'0')
}
```

```
Decimal: 63              ; decimal representation of 63
Hex: 3F                  ; representation of 63
Binary: 111111           ; binary representation of 63
Character: ?             ; 63 is the ASCII code of 63
String: ?%!c(MISSING)    ; explicit Error
String: ═╚               ; implicit Error
String: 63               ; Output ‘6’, ‘3’
```

And escape values

Value	Meaning	Example
\\	Backslash	
\t	Tab	
\n	Newline	
\"	Double quote	

```
fmt.Printf("\t Use \\\" to output %c%c\n",92,34)
```

Outputs

```
        Use \" to output \"
>
```

And escape values

Given fmt.Printf("\t Use \\\" to output %c%c\n",92,34)

The output is

```
        Use \" to output \"             // has a tab and a newline
>
```

- Delete escape values \t and \n

fmt.Printf("Use \\\" to output %c%c",92,34)

The output becomes

```
Use \" to output \">                    // missing a tab and a newline
```

And three default I/O devices

- Standard Input
  - Keyboard
  - StdIn, stdin
- Standard Output
  - Display screen
  - StdOut, stdout
- Standard Error Output
  - Display screen
  - StdErr, stderr

```
package main   // smbols.go
import "fmt"
func main() {
   fmt.Printf("Decimal: %d\n",63)
   fmt.Printf("Hex: %X\n",63)
   fmt.Printf("Binary: %b\n",63)
   fmt.Printf("Character: %c\n",63)
   fmt.Printf("String: %c%c\n",63)
   fmt.Printf("String: %c%c\n",6,3)
   fmt.Printf("String: %c%c\n",6+'0',3+'0')
}
```

```
Decimal: 63                ; binary representation of 63
Hex: 3F                    ; representation of 63
Binary: 111111             ; binary representation of 63
Character: ?               ; 63 is the ASCII code of 63
String: ?%!c(MISSING)      ; explicit Error
String: ═╚                 ; implicit Error
String: 63                 ; Output '6', '3'
```

3.3 Convert strings to numbers

- Use two examples
  - Add up the ASCII codes of a student's name string
  - name_to_number-0.go, a simple version using %d
  - name_to_number.go, using only %c
    - Implement %d with %c
- Learn variable types and for loop
  - variable types
    - Integer data type: int
    - String data type: string
    - Array data type: array
  - for loop

```
package main //name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < len(name); i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

ASCII encodings for “Alan Turing”

= [65, 108, 97, 110, 32, 84, 117, 114, 105, 110, 103]

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

name_to_number-0.go

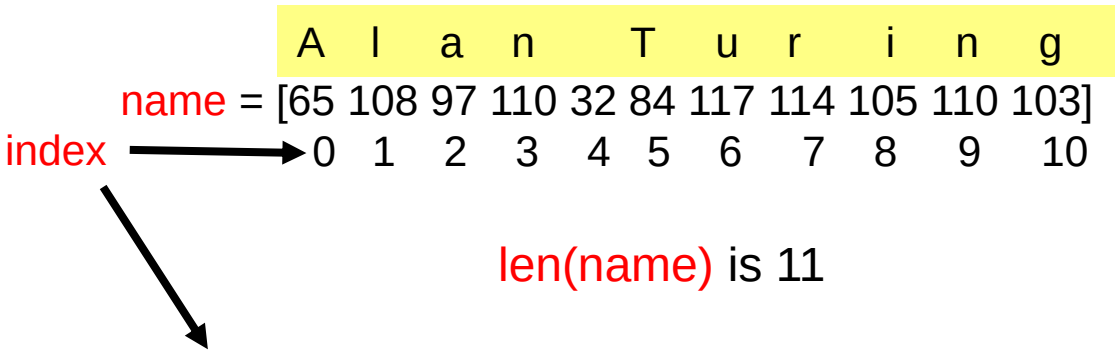
- Two ways to declare a variable

```
var sum int = 0

sum := 0
```

- String as an array of characters

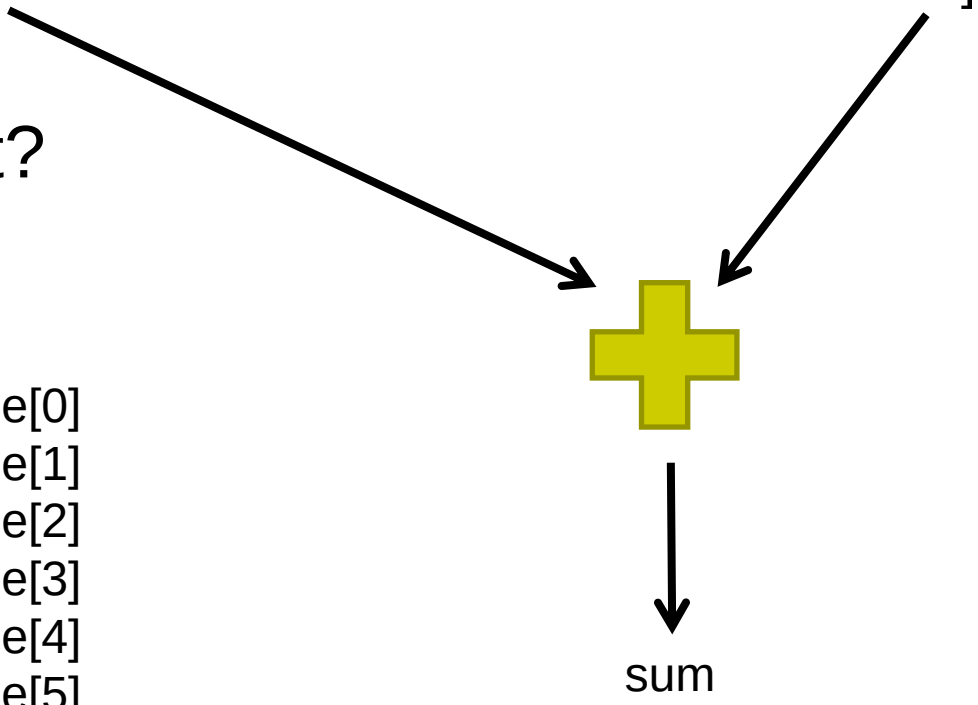
```
var name string = "Alan Turing"

name := "Alan Turing“

Note that name[4] is ‘ ’=32
```

```
package main
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < len(name); i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```



How to add up elements of an array?

- Problem: add up the 11 elements of name[i]
  - name = [65 108 97 110 32 84 117 114 105 110 103]



- How to do it?
  - Solution 1

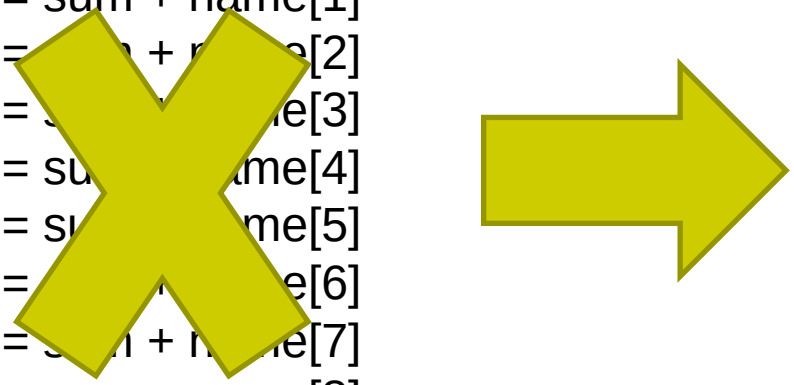
```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
sum = sum + name[2]
sum = sum + name[3]
sum = sum + name[4]
sum = sum + name[5]
sum = sum + name[6]
sum = sum + name[7]
sum = sum + name[8]
sum = sum + name[9]
sum = sum + name[10]
```

How to add up elements of an array?

- Problem: add up the 11 elements of name[i]
  - name = [65 108 97 110 32 84 117 114 105 110 103]

- How to do it?
  - Solution 1

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
sum = sum + name[2]
sum = sum + name[3]
sum = sum + name[4]
sum = sum + name[5]
sum = sum + name[6]
sum = sum + name[7]
sum = sum + name[8]
sum = sum + name[9]
sum = sum + name[10]
```



Solution 2 Why?

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
sum = sum + int(name[2])
sum = sum + int(name[3])
sum = sum + int(name[4])
sum = sum + int(name[5])
sum = sum + int(name[6])
sum = sum + int(name[7])
sum = sum + int(name[8])
sum = sum + int(name[9])
sum = sum + int(name[10])
```

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[0] = 65 = 01000001

Value Name	Binary representation
sum (right-side)	0000000000000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	0000000000000000000000000000000000000000000000000000000001000001
sum (left-side)	0000000000000000000000000000000000000000000000000000000001000001

var name string = "Alan Turing“
var sum int = 0

Type of name: an array of byte, i.e., uint8
Type of sum: 64-bit integer

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
…
sum = sum + name[10]
```

Solution 1

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
…
sum = sum + int(name[10])
```

Solution 2

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[0] = 65 = 01000001

Value Name	Binary representation
sum (right-side)	00000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	00000000000000000000000000000000000000000000000001000001
sum (left-side)	00000000000000000000000000000000000000000000000001000001

Type casting operation int(name[0]) converts byte type to int type
By padding 56 0's

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
…
sum = sum + name[10]
```

Solution 1

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
…
sum = sum + int(name[10])
```

Solution 2

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[0] = 65 = 01000001

Value Name	Binary representation
sum (right-side)	00000000000000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	00000000000000000000000000000000000000000000000000000001000001
sum (left-side)	00000000000000000000000000000000000000000000000000000001000001

sum := 0
sum = sum + int(name[0])
= 00000000000000000000000000000000000000000000000000000001000001
sum = sum + int(name[1])
=?
…
sum = sum + int(name[10])

Solution 2

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[1] = 108 = 01000001

Value Name	Binary representation
sum (right-side)	00000000000000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	00000000000000000000000000000000000000000000000000000001000001
sum (left-side)	00000000000000000000000000000000000000000000000000000001000001

sum := 0
sum = sum + int(name[0])
= 0 + 65 = 65
= 00000000000000000000000000000000000000000000000000000001000001
sum = sum + int(name[1])
= 65 + 108 = 173
= 00000000000000000000000000000000000000000000000000000010101101

Code for solution 2 still has problems

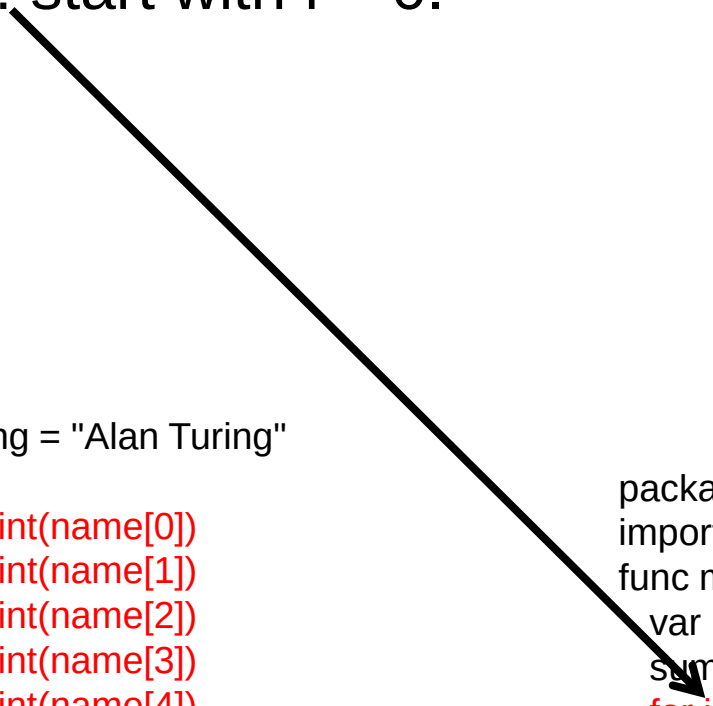
1. Tied to particular students
   - Does not work for students with len(name) != 11
2. Tedious, repetitive code
3. The length of code is proportional to the problem size!
   - **A sign of possible bad design**
     - What if len(name) == 1 million?
   - Programs should have finite lengths

- Recall Project 1: Turing Adder
  - We don't want the size of transition table of Turing machine to be proportional to input length N

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
sum = sum + int(name[2])
sum = sum + int(name[3])
sum = sum + int(name[4])
sum = sum + int(name[5])
sum = sum + int(name[6])
sum = sum + int(name[7])
sum = sum + int(name[8])
sum = sum + int(name[9])
sum = sum + int(name[10])
```

For loop comes to the rescue

- Initialize: start with i = 0.



```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   sum = sum + int(name[0])
   sum = sum + int(name[1])
   sum = sum + int(name[2])
   sum = sum + int(name[3])
   sum = sum + int(name[4])
   sum = sum + int(name[5])
   sum = sum + int(name[6])
   sum = sum + int(name[7])
   sum = sum + int(name[8])
   sum = sum + int(name[9])
   sum = sum + int(name[10])
   fmt.Printf("%d\n", sum)
}
```

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   fmt.Printf("%d\n", sum)
}
```

For loop comes to the rescue

- Initialize: start with i = 0.
- Repetitively execute the loop body until i≥11.

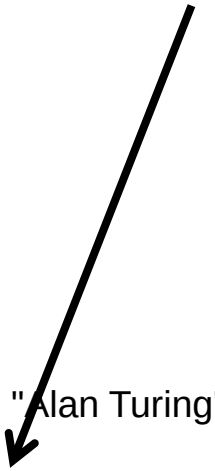
```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   sum = sum + int(name[0])
   sum = sum + int(name[1])
   sum = sum + int(name[2])
   sum = sum + int(name[3])
   sum = sum + int(name[4])
   sum = sum + int(name[5])
   sum = sum + int(name[6])
   sum = sum + int(name[7])
   sum = sum + int(name[8])
   sum = sum + int(name[9])
   sum = sum + int(name[10])
   fmt.Printf("%d\n", sum)
}
```

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   fmt.Printf("%d\n", sum)
}
```

For loop comes to the rescue

- Initialize: start with i = 0.
- Repetitively execute the loop body until i≥11.
- After each repetition (called iteration), increment i by 1.

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   sum = sum + int(name[0])
   sum = sum + int(name[1])
   sum = sum + int(name[2])
   sum = sum + int(name[3])
   sum = sum + int(name[4])
   sum = sum + int(name[5])
   sum = sum + int(name[6])
   sum = sum + int(name[7])
   sum = sum + int(name[8])
   sum = sum + int(name[9])
   sum = sum + int(name[10])
   fmt.Printf("%d\n", sum)
}
```



```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   fmt.Printf("%d\n", sum)
}
```

3.3 Use verb %c to implement verb %d

name_to_number-0.go
vs.
name_to_number.go

```
package main //name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < 11; i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

```
> ./name_to_number-0
> 1045
>
```

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum != 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

```
> ./name_to_number
> 1045
>
```

How to implement %d with %c?

How to implement %d with %c?

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum != 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

Use / and % to extract 5, 4, 0, 1 from 1045

sum % 10
sum = sum / 10

sum_bytes[3]	1045 % 10 = 5
sum	1045 / 10 = 104
sum_bytes[2]	104 % 10 = 4
sum	104 / 10 = 10
sum_bytes[1]	10 % 10 = 0
sum	10 / 10 = 1
sum_byte[0]	1 % 10 = 1
sum	1 / 10 = 0

sum_bytes = [‘1’, ‘0’, ‘4’, ‘5’]
0 1 2 3

Use character verb %c to
Implement decimal verb %d
fmt.Printf("%d\n", sum)

The four fmt.Printf("%c", …) statements output 1, 0, 4, 5 one by one

Why type casting?

In sum_bytes[j] = byte(sum%10) + '0'

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum != 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

Why not using
sum_bytes[j] = sum % 10

sum_bytes is a uint8 array
sum_byte[j] has type byte
However, sum is type int

Suppose j=3 (initially)
sum is 1045, and
sum%10 is
1045%10 = 5, a 64-bit int value
00000000……00000101

byte(sum%10), i.e., byte(5)
converts this 64-bit value
to a number of type uint8 and value
00000101

byte(5)+ ‘0’ evaluates to
= 5 + 48
= 00000101+00110000
= 00110101 = 53 = ‘5’

Thus, sum_bytes[3] holds ‘5’

3.5 Good Programming Practices

- The code of name_to_number.go shows bad programming practices
  - Non-descriptive names
    - name, sum_bytes
  - Magic numbers
    - 11, 4, 3
  - Repetitive code
    - of four print statements
  - No documentation

```
package main
import "fmt"
func main() {
var name string = "Alan Turing"
    sum := 0
    for i := 0; i < 11; i++ {
       sum = sum + int(name[i])
    }
var sum_bytes [4]byte
var j int
for j = 3; sum != 0; j-- {
   sum_bytes[j] = byte(sum%10) + '0'
   sum = sum / 10
}
fmt.Printf("%c", sum_bytes[0])
fmt.Printf("%c", sum_bytes[1])
fmt.Printf("%c", sum_bytes[2])
fmt.Printf("%c", sum_bytes[3])
fmt.Printf("\n")
}
```

name_to_number-1.go resolves these issues

- Non-descriptive names
  - name, sum_bytes
  - studentName
  - maxCodeLength
  - sumBytes
- Magic numbers
  - 11 → len(studentName)
  - 4 → maxCodeLength
  - 3 → len(sumBytes) - 1
- Repetitive code
  - for k loop
- No documentation
  - 5 lines of comments added

```
package main
import "fmt"
const studentName       = "Alan Turing"
const maxCodeLength     = 4 // student code has at most 4 digits
func main() {
    sum := 0
    for i := 0; i < len(studentName); i++ { // add up studentName to sum
       sum = sum + int(studentName[i])
    }
   var sumBytes [maxCodeLength]byte  // array to hold characters of sum
   var j int
   for j = len(sumBytes) - 1; sum != 0; j-- { // extract each digit from sum
      sumBytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   var k int
   for k = j + 1; k < len(sumBytes); k++ { // print each digit of sum
      fmt.Printf("%c", sumBytes[k])
   }
   fmt.Printf("\n")
}
```

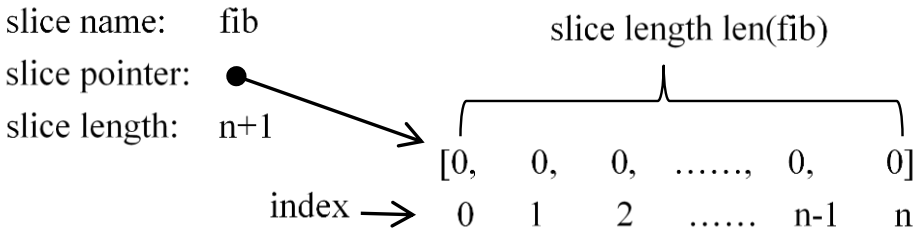
3.6 A Fibonacci program using slice

- Program fib.dp.go uses a slice fib to store F(i), i = 0, 1, …, n
  - n = 50 in this example
- A slice is dynamically created by a system-provided make function, working for any n
  - In contrast, an array (or a slice) has a fixed length which is part of its type
- A slice is a structure pointing to an underlying array
- Dynamic programming: Every newly computed Fibonacci value is stored (memorized) in slice element fib[i] and later referenced. No Fibonacci value is computed more than once.

Program fib.dp-50.go

```
package main
import "fmt"
func main() {
   fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1) // make a slice fib
   fib[0] = 0                       // initialize fib[0] and fib[1]
   fib[1] = 1
   for i := 2; i <= n; i++ {        // iteratively compute fib[i]
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2]	…	fib[50]
After …= make([]int, n+1)	0	0	0		0
…					

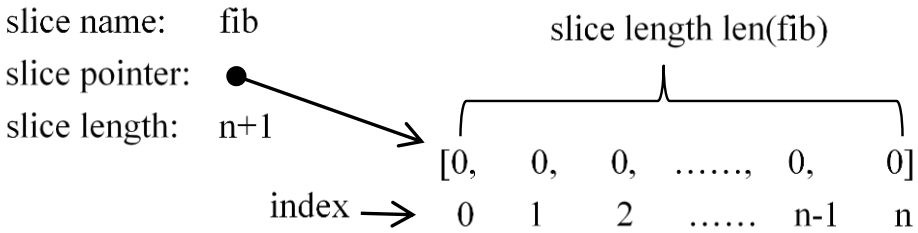
3.6 A Fibonacci program using slice

- Program fib.dp.go uses a slice fib to store F(i), i = 0, 1, …, n
  - n = 50 in this example
- A slice is dynamically created by a system-provided make function, working for any n
  - In contrast, an array (or a slice) has a fixed length which is part of its type
- A slice is a structure pointing to an underlying array
- Dynamic programming: Every newly computed Fibonacci value is stored (memorized) in slice element fib[i] and later referenced. No Fibonacci value is computed more than once.

Program fib.dp-50.go

```
package main
import "fmt"
func main() {
   fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1) // make a slice fib
   fib[0] = 0                       // initialize fib[0] and fib[1]
   fib[1] = 1
   for i := 2; i <= n; i++ {        // iteratively compute fib[i]
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2]	…	fib[50]
After …= make([]int, n+1)	0	0	0		0
After fib[0] = 0	0	0	0		0

…

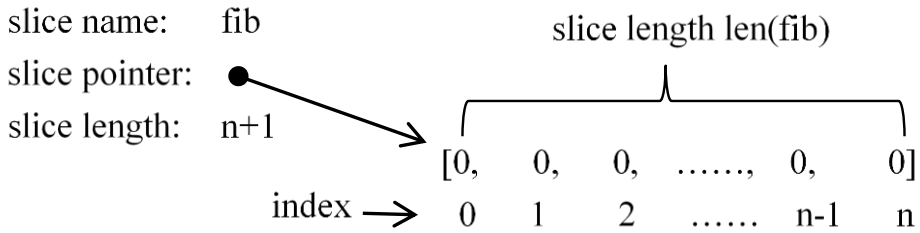
3.6 A Fibonacci program using slice

- Program fib.dp.go uses a slice fib to store F(i), i = 0, 1, …, n
  - n = 50 in this example
- A slice is dynamically created by a system-provided make function, working for any n
  - In contrast, an array (or a slice) has a fixed length which is part of its type
- A slice is a structure pointing to an underlying array
- Dynamic programming: Every newly computed Fibonacci value is stored (memorized) in slice element fib[i] and later referenced. No Fibonacci value is computed more than once.

Program fib.dp-50.go

```
package main
import "fmt"
func main() {
   fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1) // make a slice fib
   fib[0] = 0                       // initialize fib[0] and fib[1]
   fib[1] = 1
   for i := 2; i <= n; i++ {        // iteratively compute fib[i]
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2]	…	fib[50]
After …= make([]int, n+1)	0	0	0		0
After fib[0] = 0	0	0	0		0
After fib[1] = 1	0	1	0		0

…

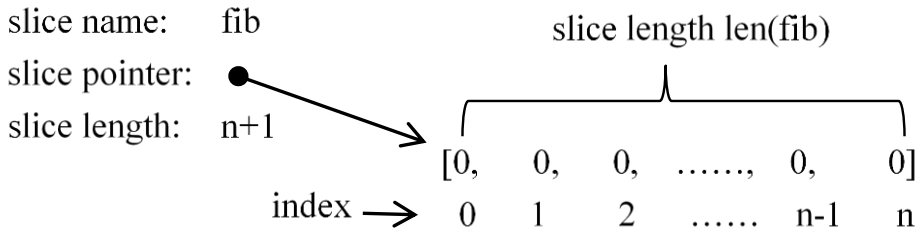
3.6 A Fibonacci program using slice

- Program fib.dp.go uses a slice fib to store F(i), i = 0, 1, …, n
  - n = 50 in this example
- A slice is dynamically created by a system-provided make function, working for any n
  - In contrast, an array (or a slice) has a fixed length which is part of its type
- A slice is a structure pointing to an underlying array
- Dynamic programming: Every newly computed Fibonacci value is stored (memorized) in slice element fib[i] and later referenced. No Fibonacci value is computed more than once.

Program fib.dp-50.go

```
package main
import "fmt"
func main() {
   fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   var fib []int = make([]int, n+1) // make a slice fib
   fib[0] = 0                       // initialize fib[0] and fib[1]
   fib[1] = 1
   for i := 2; i <= n; i++ {        // iteratively compute fib[i]
      fib[i] = fib[i-1] + fib[i-2]
   }
   return fib[n]
}
```

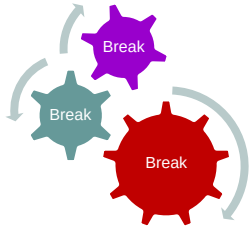
var fib []int = make([]int, n+1)



	fib[0]	fib[1]	fib[2]	…	fib[50]
After …= make([]int, n+1)	0	0	0		0
After fib[0] = 0	0	0	0		0
After fib[1] = 1	0	1	0		0
After fib[2] = fib[1]+fib[0]	0	1	1		0

…

Take-Home Messages



- Introducing abstractions: program constructs
  - Variables, constants, expressions
    - byte (uint8), int, uint; array, string, slice
  - Statements
    - package, import, assignment, sequence, if-then-else, for loop
  - Others
    - Formatting verbs, escape values
    - Function definition and call; function can have side effect
    - Built-in function: make a slice
- Introducing abstracting
  - Packaging,
  - Good programming practices
  - Dynamic programming
    - Memorize intermediate results to avoid redundant computations, and can significantly speed up programs