More Examples

02-201 / 02-601

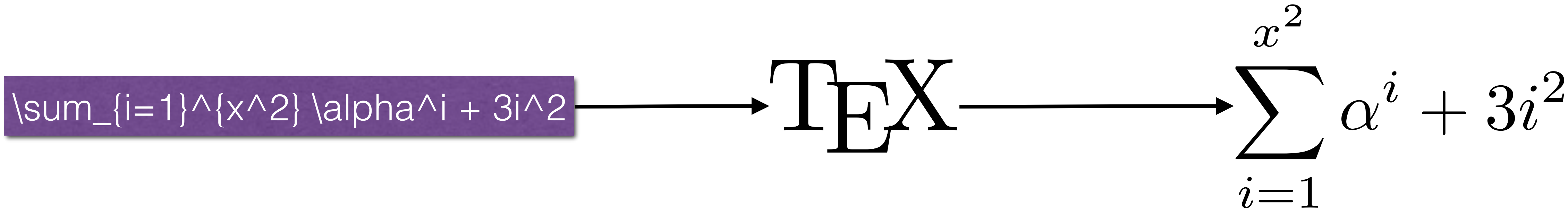
Debugging

How to Debug

- Debugging is solving a murder mystery: you see the evidence of some failure & you try to figure out what part of the code caused it.

- It’s based on backward reasoning:
  - I see A, what could have caused A?
  - It could have been B or C or D or E or F.
  - It isn’t B because if I comment out that code, I still get the problem.
  - It isn’t C because if I add an “if” statement to check if C is happening, I see that it is not.
  - It isn’t D because I wrote a small test program, and D can’t happen.
  - It isn’t E because I print out the value of E, and it’s correct.
  - So it must be F.

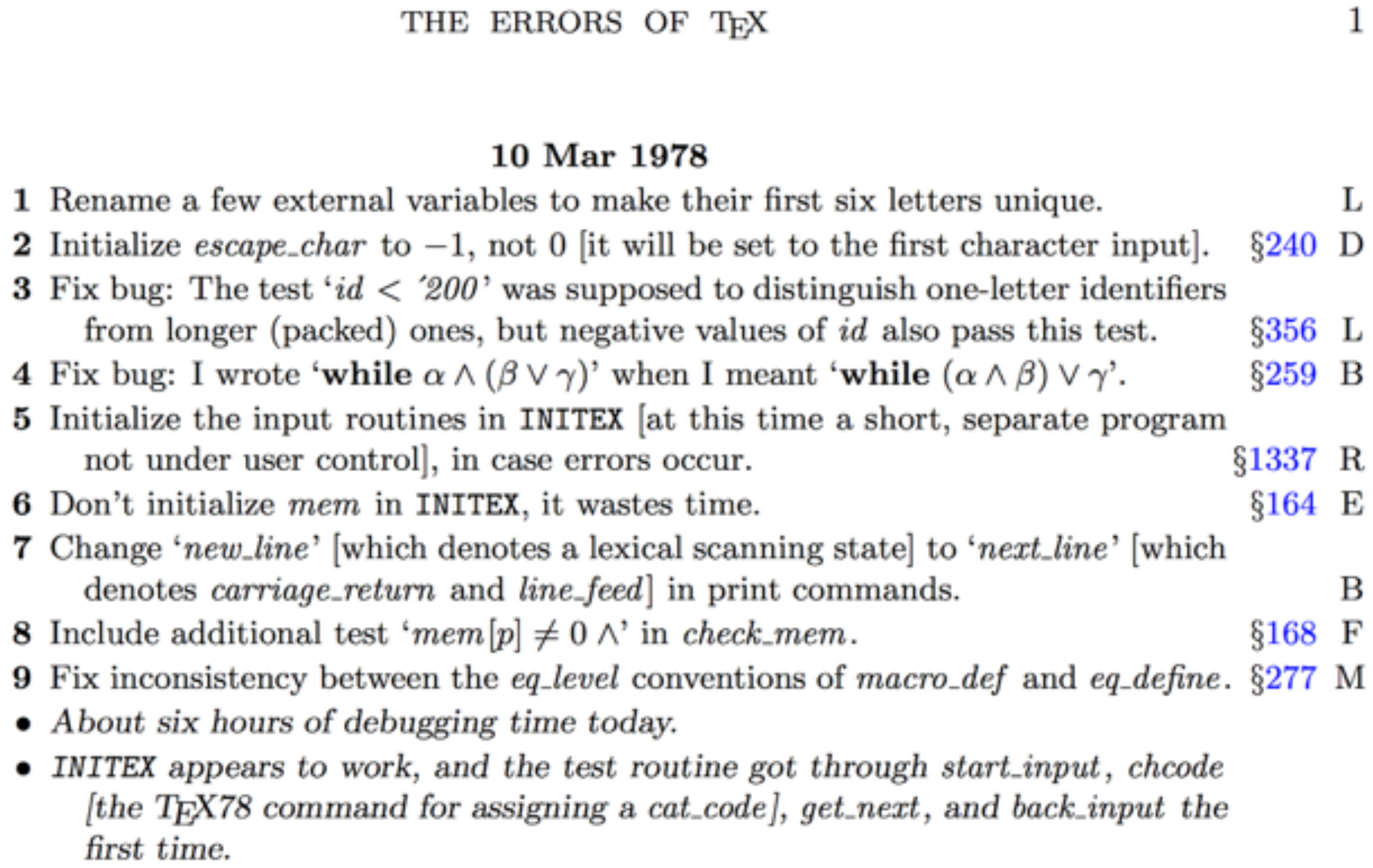
Bug are "normal"



```
\sum_{i=1}^{x^2} \alpha^i + 3i^2
```

$$\sum_{i=1}^{x^2} \alpha^i + 3i^2$$

TeX is a system for typesetting mathematical and scientific papers.
It's was written by Don Knuth (Stanford CS prof), and still widely used.



THE ERRORS OF TEX 1

10 Mar 1978

1 Rename a few external variables to make their first six letters unique. L

2 Initialize *escape_char* to −1, not 0 [it will be set to the first character input]. §240 D

3 Fix bug: The test '*id* < '*200*' was supposed to distinguish one-letter identifiers from longer (packed) ones, but negative values of *id* also pass this test. §356 L

4 Fix bug: I wrote '**while** $\alpha \wedge (\beta \vee \gamma)$' when I meant '**while** $(\alpha \wedge \beta) \vee \gamma$'. §259 B

5 Initialize the input routines in `INITEX` [at this time a short, separate program not under user control], in case errors occur. §1337 R

6 Don't initialize *mem* in `INITEX`, it wastes time. §164 E

7 Change '*new_line*' [which denotes a lexical scanning state] to '*next_line*' [which denotes *carriage_return* and *line_feed*] in print commands. B

8 Include additional test '$mem[p] \neq 0 \wedge$' in *check_mem*. §168 F

9 Fix inconsistency between the *eq_level* conventions of *macro_def* and *eq_define*. §277 M

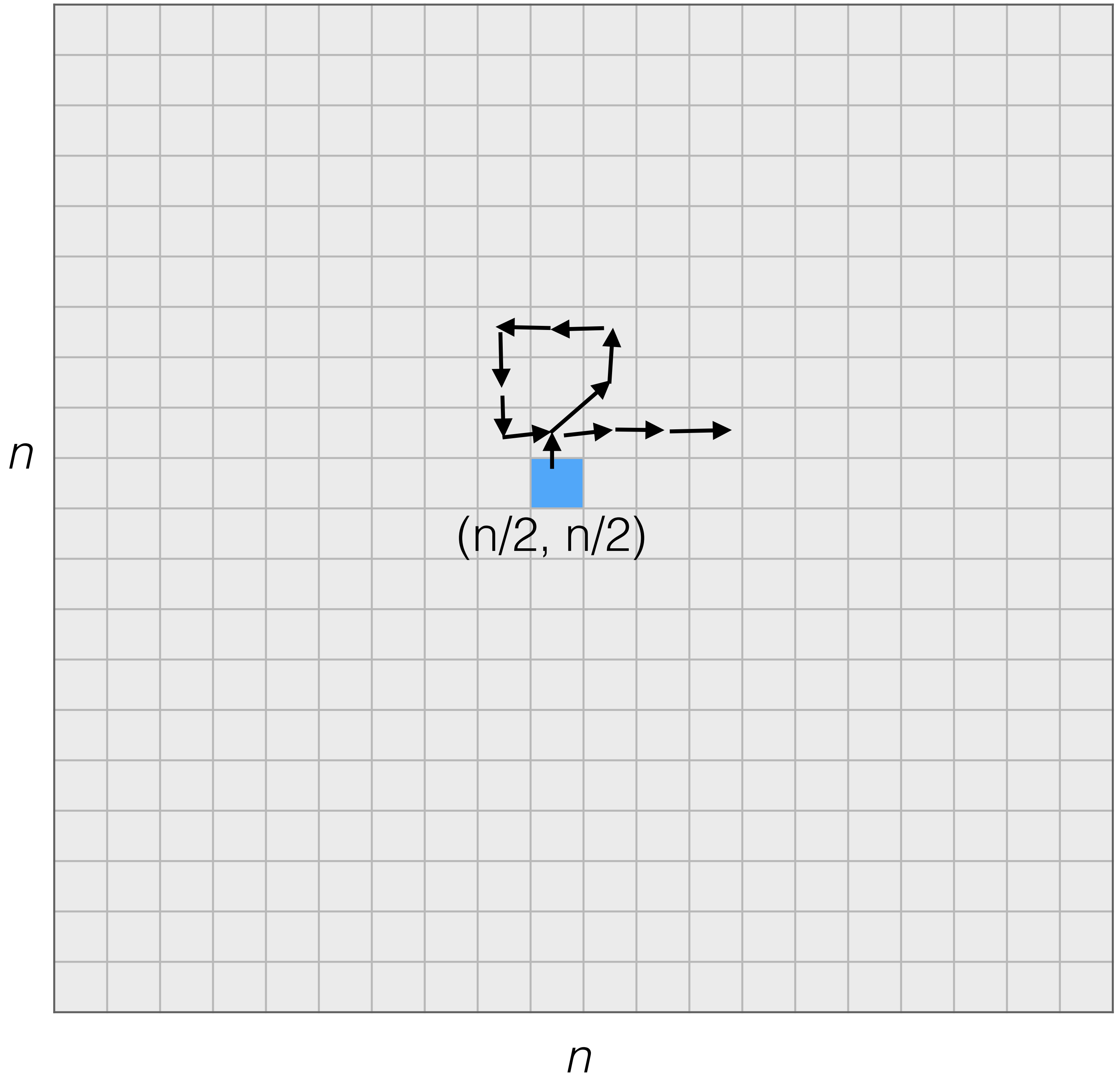
- *About six hours of debugging time today.*
- *INITEX appears to work, and the test routine got through start_input, chcode [the TEX78 command for assigning a cat_code], get_next, and back_input the first time.*

http://texdoc.net/texmf-dist/doc/generic/knuth/errata/errorlog.pdf

More Programming Examples

Example: Random Walks

Simulate a random walk on an n-by-n chessboard



Example: Random Walks

Simulate a random walk on an n-by-n chessboard

```go
func randDelta() int {
    return (rand.Int() % 3) - 1
}

func randomWalk(n, steps int) {
    var x, y = n/2, n/2
    fmt.Println(x,y)
    for i := 0; i < steps; i++ {
        var dx, dy int

        for dx == 0 && dy == 0 {
            dx = randDelta()
            for  x+dx < 0 || x+dx >= n {
                dx = randDelta()
            }

            dy = randDelta()
            for y+dy < 0 || y+dy >= n {
                dy = randDelta()
            }
        }
        x += dx
        y += dy
        fmt.Println(x,y)
    }
}
```

rand.Int() returns a random non-negative integer.

Must put

import “math/rand”

at top of your program.

Loop to make sure we move.

Loop to keep position within [0, n) x [0, n)

Note the code duplicating the test for an in-field coordinate.

This isn’t very good.

Better to break this out into a function.

```
x y
5 5
4 5
3 4
2 5
3 4
4 5
5 6
4 5
3 4
4 4
5 4
4 5
4 6
4 5
4 6
5 7
6 8
5 8
5 7
6 6
```

New Version With Better Functions

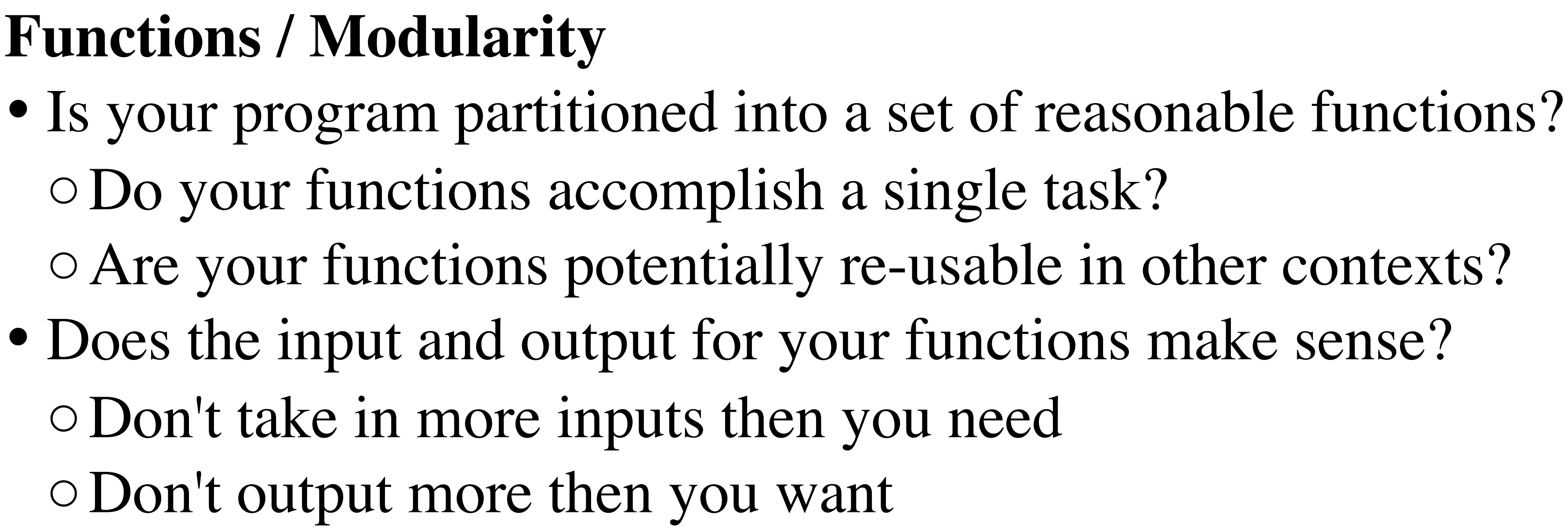
```
func randDelta() int {
    return (rand.Int() % 3) - 1
}

func inField(coord, n int) bool {
    return coord >= 0 && coord < n
}

func randStep(x,y,n int) (int, int) {
    var nx, ny int = x, y
    for (nx == x && ny == y) || !inField(nx,n) || !inField(ny,n) {
       nx = x+randDelta()
       ny = y+randDelta()
    }
    return nx, ny
}

func randomWalk(n, steps int) {
    var x, y = n/2, n/2
    fmt.Println(x,y)
    for i := 0; i < steps; i++ {
        x,y = randStep(x,y,n)
        fmt.Println(x,y)
    }
}
```

This version is:

- clearer
- more flexible — perhaps we can use randStep() someplace else.
- Slightly shorter (25 vs. 26 lines)

“Style” Tip

- Break your program into short functions that do a single, well-defined job.



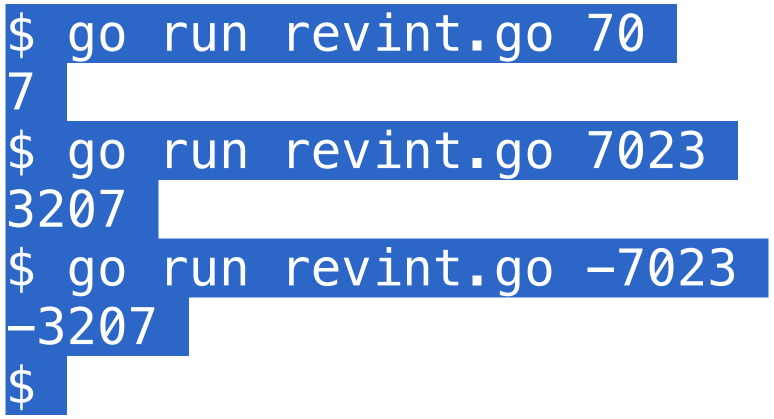
Functions / Modularity

- Is your program partitioned into a set of reasonable functions?
  - Do your functions accomplish a single task?
  - Are your functions potentially re-usable in other contexts?
- Does the input and output for your functions make sense?
  - Don't take in more inputs then you need
  - Don't output more then you want

- Not just about “style”, but small functions let you think about one thing at a time.

Command Line Arguments

Command line arguments provide a way for the user to pass values into your program:



```
$ go run revint.go 70
7
$ go run revint.go 7023
3207
$ go run revint.go -7023
-3207
$
```

Package “os” provides access to these parameters:

`os.Args[i]` the *i*th argument on the command line when your program was run. (*i*=0 is a special case and the parameters are in *i* ≥ 1)

`len(os.Args)` the number of command line arguments **+ 1**

```
import “os”

    // …

func main() {
    fmt.Println(os.Args[1])
}
```

```
go run myProgram.go a 3 77 “another param”
```

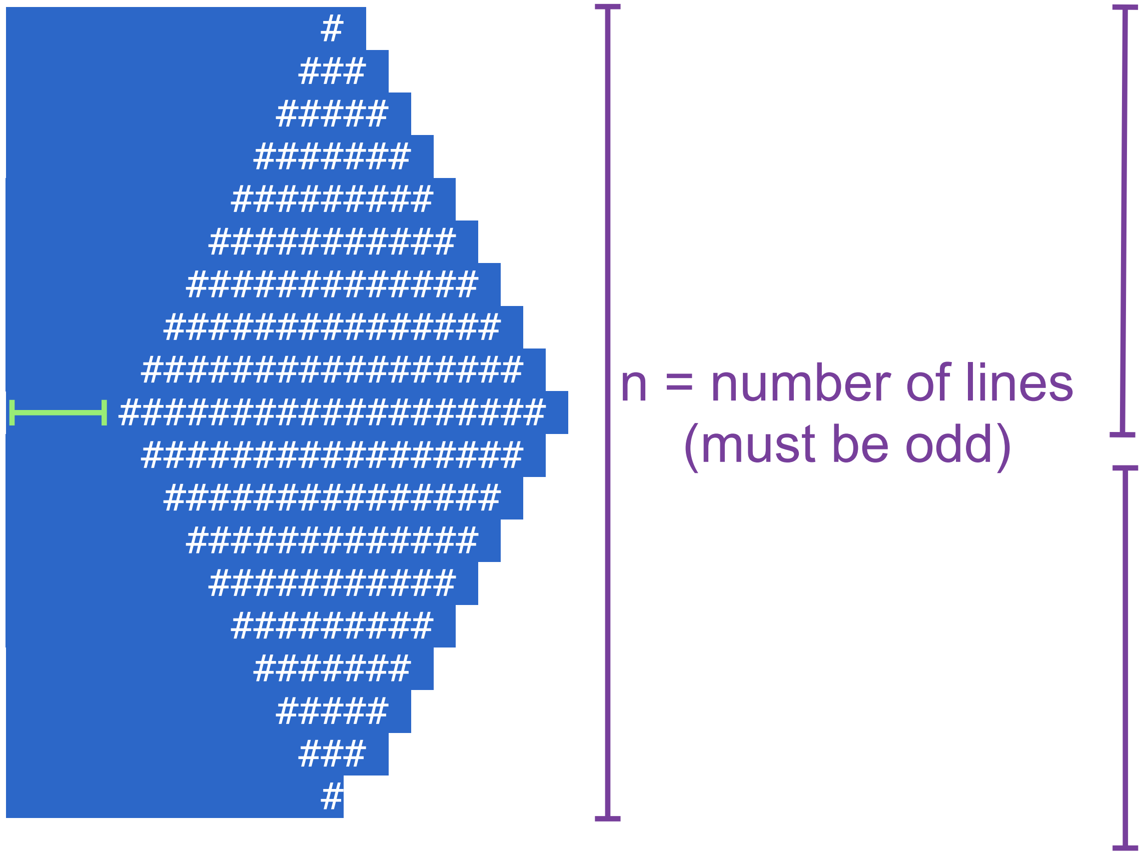
- `os.Args[1] = “a”`
- `os.Args[2] = “3”`
- `os.Args[3] = “77”`
- `os.Args[4] = “another param”`
- `os.Args[0] = “myProgram”`

Note: all os.Args[i] are ***strings*** even if they look like numbers.

os.Args[0] holds some representation of the *name* of your program.

Example: Print a Diamond

```
func printDiamond(n, shift int)
```



shift = number of characters to shift diamond right

```
              #
             ###
            #####
           #######
          #########
         ###########
        #############
       ###############
      #################
     ###################
      #################
       ###############
        #############
         ###########
          #########
           #######
            #####
             ###
              #
```

```
printDiamond(19,5)
```

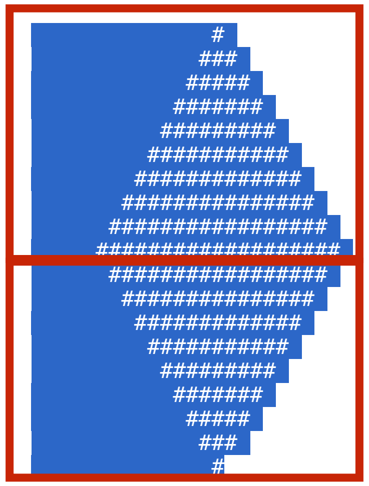
n = number of lines (must be odd)

$\lceil n/2 \rceil$ ceil = largest integer ≤ n / 2

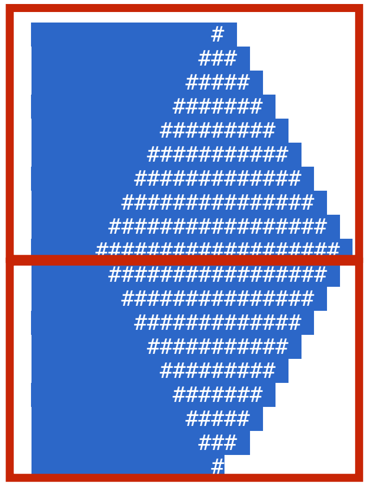
$\lfloor n/2 \rfloor$ floor = smallest integer ≥ n / 2

Break into two subproblems:

```
    printTriangle(n, shift int)
    printInvertedTriangle(n, shift int)
```



Example: printDiamond



```
         #
        ###
       #####
      #######
     #########
    ###########
   #############
  ###############
 #################
###################
 #################
  ###############
   #############
    ###########
     #########
      #######
       #####
        ###
         #
```

Break into two subproblems:

```
    printTriangle(n, shift int)
    printInvertedTriangle(n, shift int)
```

```
func printDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        printTriangle(n / 2 + 1, shift)
        printInvertedTriangle(n/2, shift+1)
    }
}
```

Check that the parameters are valid. This is good practice.

Print top triangle.

Print bottom triangle.

Since *n* is odd:

$$\lceil n/2 \rceil = n/2 + 1$$

$$\lfloor n/2 \rfloor = n/2$$

What's going on here?
Since n is an integer variable and 2 is an integer
the code `n / 2` does ***integer*** division and rounds down.

The bottom triangle is slightly shorter and shifted to the right by 1 extra space.

Top-Down Program Design

- We “used” the `printTriangle()` and `printInvertedTriangle()` functions in our thinking before we wrote them.

- We know what they are supposed to do, so we could use them to write `printDiamond()` even before we implemented them.

- In a sense, it doesn’t matter *how* `printTriangle()` and `printInvertedTriangle()` are implemented: if they do what they are supposed to do, everything will work.

- It’s only their ***interface*** to the rest of the program that matters.

- This is top-down design, and it’s often a very good way to approach writing programs:

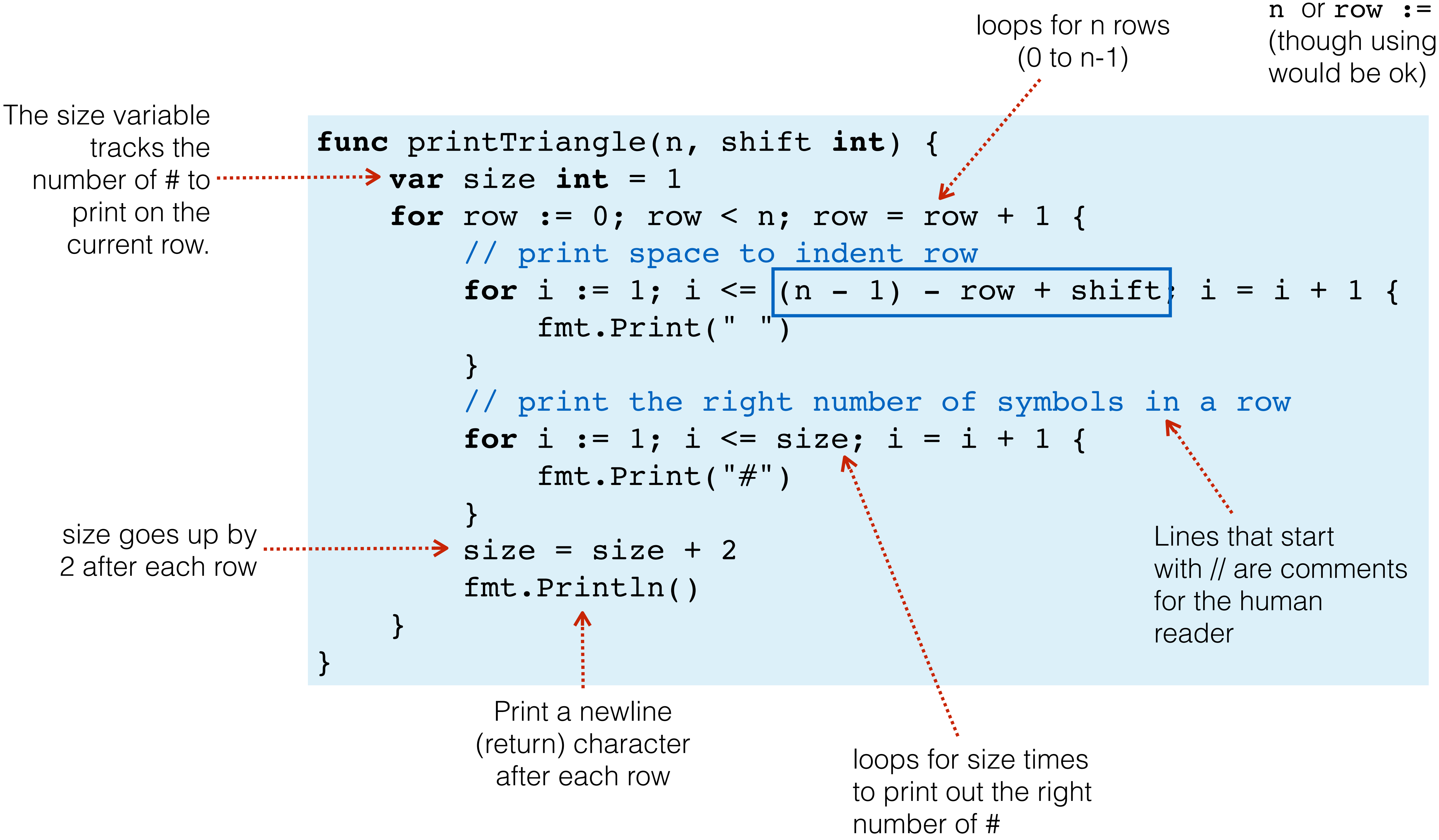
    1. start by breaking down your task into subproblems.
    2. write a solution to the top-most subproblem using functions for other subproblems that you will write later.
    3. then repeat by writing solutions to those subproblems, possibly breaking *them* up into subproblems.

Good Programming:

Break big problems into small functions with good interfaces.

printTriangle(n,shift)

Tip: watch out for “off-by-one” errors: e.g. using `row <= n` or `row := 1` (though using both would be ok)



```
func printTriangle(n, shift int) {
    var size int = 1
    for row := 0; row < n; row = row + 1 {
        // print space to indent row
        for i := 1; i <= (n - 1) - row + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size + 2
        fmt.Println()
    }
}
```

loops for n rows (0 to n-1)

The size variable tracks the number of # to print on the current row.

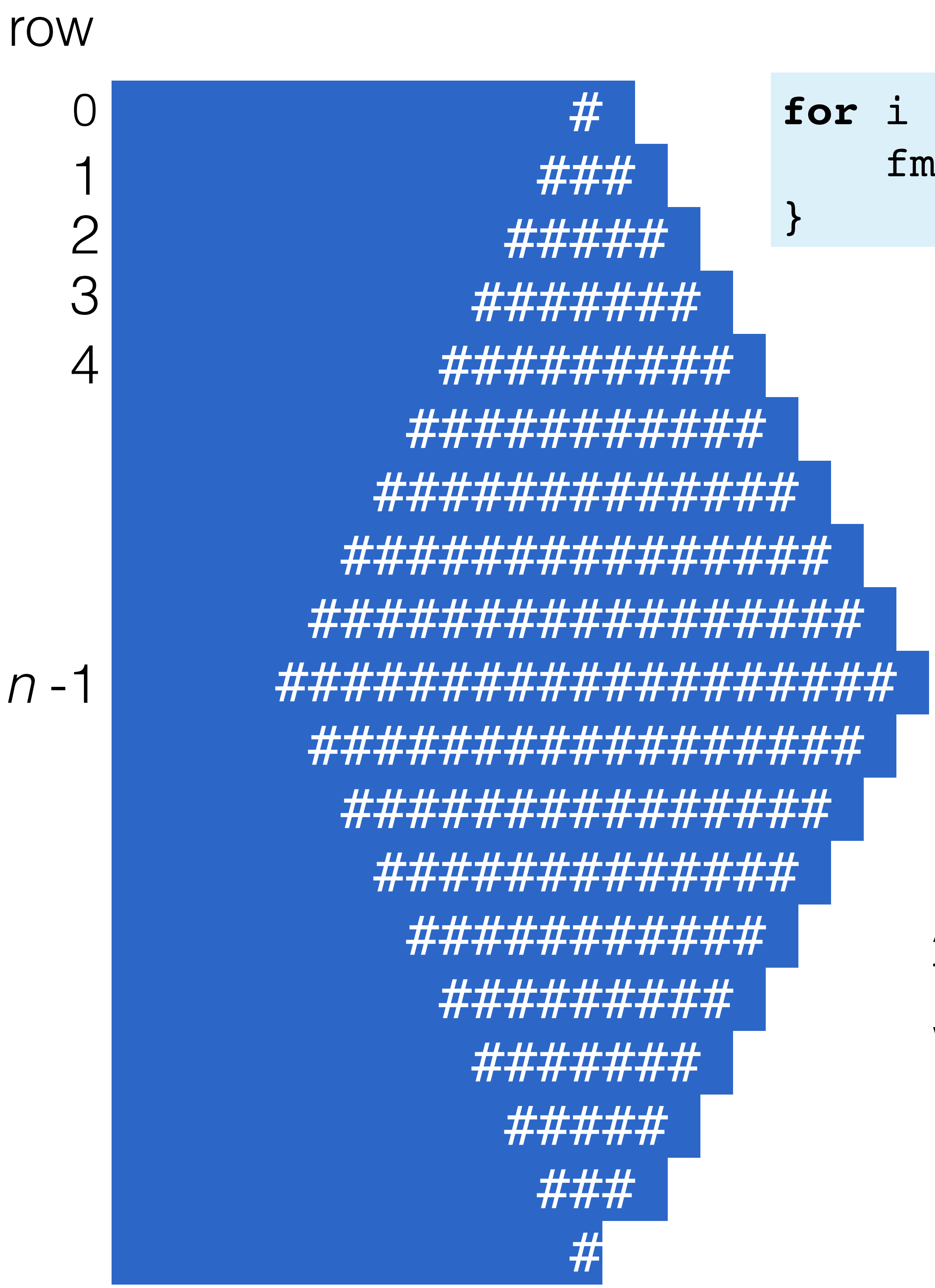
size goes up by 2 after each row

Lines that start with // are comments for the human reader

Print a newline (return) character after each row

loops for size times to print out the right number of #

Why n - row - 1 + shift?



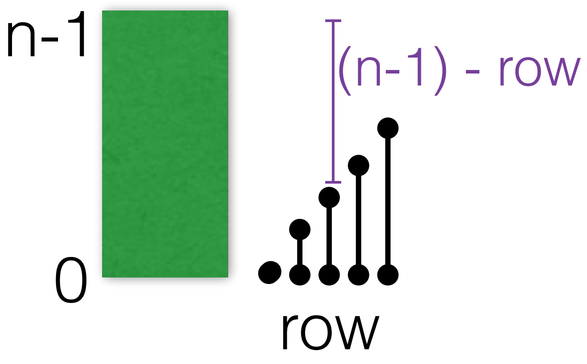
```
for i := 1; i <= (n - 1) - row + shift; i = i + 1 {
    fmt.Print(" ")
}
```

when row = n-3, loop should execute 2 + shift times

when row = n-2, loop should execute 1 + shift times

when row = n-1, loop should execute shift times

At each row, one fewer space should be written.
The last row (numbered n-1) should have shift spaces written.



printInvertedTriangle(n,shift)

size starts at the size of the top-most row, which has 2n - 1 symbols in it.

In first iteration of the row loop, row == n, so n - row = 0, and this loop iterates shift times

```
func printInvertedTriangle(n, shift int) {
    var size int = 2*n - 1
    // Note: this loop counts down
    for row := n; row > 0; row = row - 1 {
        for i := 1; i <= n - row + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }

        size = size - 2
        fmt.Println()
    }
}
```

Complete Code for Diamond Example

```go
func printTriangle(n, shift int) {
    var size int = 1
    for row := 0; row < n; row = row + 1 {
        // print space to indent row
        for i := 1; i <= n - row - 1 + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size + 2
        fmt.Println()
    }
}

func printInvertedTriangle(n, shift int) {
    var size int = 2*n - 1
    // Note: this loop counts down
    for row := n; row > 0; row = row - 1 {
        for i := 1; i <= n - row + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size - 2
        fmt.Println()
    }
}

func printDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        printTriangle(n / 2 + 1, shift)
        printInvertedTriangle(n/2, shift+1)
    }
}
```

Nested statements are indented for clarity

Comments are added to make code more readable

(don’t overdo comments though!)

A worse way to write printDiamond()

```go
func badPrintDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        var size int = 1
        for row := 0; row < n/2+1; row = row + 1 {
            // print space to indent row
            for i := 1; i <= (n/2+1) - row - 1 + shift; i = i + 1 {
                fmt.Print(" ")
            }
            // print the right number of symbols in a row
            for i := 1; i <= size; i = i + 1 {
                fmt.Print("#")
            }
            size = size + 2
            fmt.Println()
        }

        size = n - 1
        for row := (n/2); row > 0; row = row - 1 {
            for i := 1; i <= (n/2) - row + shift+1; i = i + 1 {
                fmt.Print(" ")
            }
            // print the right number of symbols in a row
            for i := 1; i <= size; i = i + 1 {
                fmt.Print("#")
            }
            size = size - 2
            fmt.Println()
        }
    }
}
```

Bug! In fact, there is a subtle bug here:

Must understand the entire function before you really know what it does.
Bugs in top part affect execution of bottom part (what if you reassigned n accidentally someplace?)

Coding Style

Style #1

- Indent blocks of code: things inside of a {} should be indented and aligned.
  - Go convention is to use a TAB
  - 2 - 4 spaces is also ok.
  - But be consistent.

```
func ReverseInteger(n int) int {
    out := 0
    for n != 0 {
        out = 10*out + n % 10
        n = n / 10
    }
    return out
}
```

```
func badReverseInteger(n int) int {
out := 0
for n != 0 {
out = 10*out + n % 10
n = n / 10
}
return out}
```

- Use consistent spacing: e.g.:

```
func Hypergeometric(a,b,c,d int) int {
//…
}
```

```
func Hypergeometric(a,  b,c, d int) int {
//…
}
```

- Choose descriptive variable names:

`numSteps, numberOfSteps, nSteps`

`n`

short variable names are ok when: they are “loop variables” (like i), or they are the main variable in a short function (see ReverseInteger above)

- Don’t use the same name for two things. (e.g. don’t use a variable named KthDigit inside a function named KthDigit)

Nested loops: Printing a “Square”

```go
func printSquare(n int) {
    for i := 1; i <= n; i=i+1 {
        for j := 1; j <= n; j=j+1 {
            fmt.Print("#")
        }
        fmt.Println("")
    }
}
```

```
carlk$ go run square.go
##########
##########
##########
##########
##########
##########
##########
##########
##########
##########
```

Style #2: Comments

- Use comments to describe tricky or confusing things in your code
  - text from // to the end of a line is a comment.

- Also use comments to document what a function does.

```
// ReverseInteger(n) will return a new integer formed by
// the decimal digits of n reversed.
func ReverseInteger(n int) int {
    out := 0
    for n != 0 {
        out = 10*out + n % 10
        n = n / 10    // note: integer division!
    }
    return out
}
```

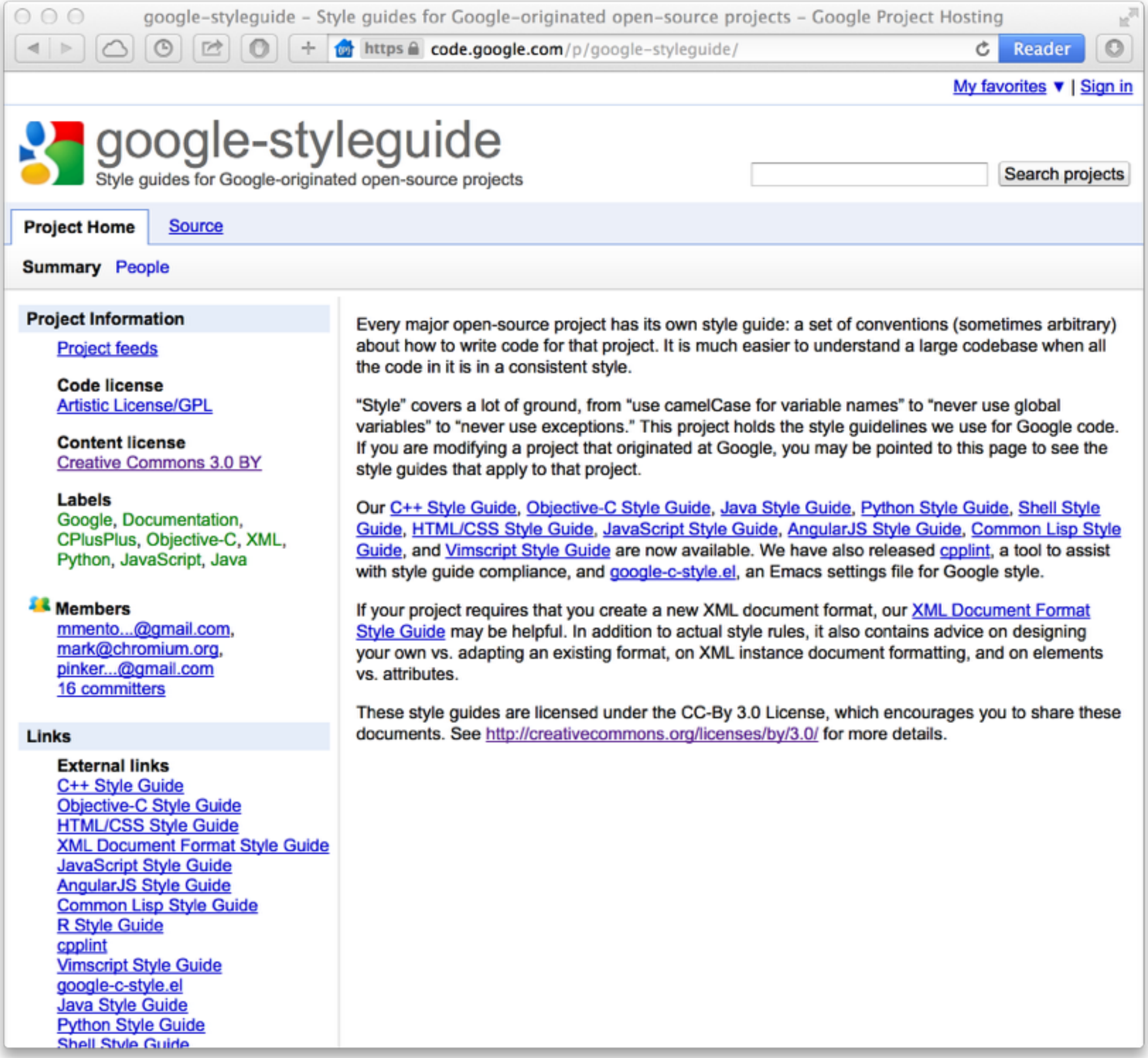
- Code between /* and */ is also a comment:

```
/* ReverseInteger(n) will return a new integer formed by
the decimal digits of n reversed. */
func ReverseInteger(n int) int {
//…
```

Goal with Style:

Readability & Consistency

Important because it’s likely you or someone else will have to modify or maintain this code later.


google-styleguide - Style guides for Google-originated open-source projects - Google Project Hosting
https code.google.com/p/google-styleguide/
Reader
My favorites ▼ | Sign in
Search projects
Project Home
Source
Summary
People

google-styleguide

Style guides for Google-originated open-source projects

Project Information

Project feeds

Code license
Artistic License/GPL

Content license
Creative Commons 3.0 BY

Labels
Google, Documentation, CPlusPlus, Objective-C, XML, Python, JavaScript, Java

Members
mmento...@gmail.com,
mark@chromium.org,
pinker...@gmail.com
16 committers

Links

External links
C++ Style Guide
Objective-C Style Guide
HTML/CSS Style Guide
XML Document Format Style Guide
JavaScript Style Guide
AngularJS Style Guide
Common Lisp Style Guide
R Style Guide
cpplint
Vimscript Style Guide
google-c-style.el
Java Style Guide
Python Style Guide
Shell Style Guide

Every major open-source project has its own style guide: a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style.

"Style" covers a lot of ground, from "use camelCase for variable names" to "never use global variables" to "never use exceptions." This project holds the style guidelines we use for Google code. If you are modifying a project that originated at Google, you may be pointed to this page to see the style guides that apply to that project.

Our C++ Style Guide, Objective-C Style Guide, Java Style Guide, Python Style Guide, Shell Style Guide, HTML/CSS Style Guide, JavaScript Style Guide, AngularJS Style Guide, Common Lisp Style Guide, and Vimscript Style Guide are now available. We have also released cpplint, a tool to assist with style guide compliance, and google-c-style.el, an Emacs settings file for Google style.

If your project requires that you create a new XML document format, our XML Document Format Style Guide may be helpful. In addition to actual style rules, it also contains advice on designing your own vs. adapting an existing format, on XML instance document formatting, and on elements vs. attributes.

These style guides are licensed under the CC-By 3.0 License, which encourages you to share these documents. See http://creativecommons.org/licenses/by/3.0/ for more details.

go fmt

- Go provides an automatic code formatting utility called “go fmt”.

- Usage:

```
$ go fmt revint.go
revint.go
```

- This will reformat your Go program using the preferred Go style, with all the correct indentations, etc.

- Note: your program must be a correct Go program for this to work (it won’t format code with syntax errors)

Style Guidelines (25% of HW grades)
02-201 / 02-601: Programming for Scientists

Variables (5 points)

- Do your variables follow proper naming convention?
  - Descriptive but not pedantic
- Do your variables fit into the proper scope?
  - Global variables are almost always bad

Functions / Modularity (10 points)

- Is your program partitioned into a set of reasonable functions?
  - Do your functions accomplish a single task?
  - Are your functions re-usable in other contexts?
- Does the input and output for your functions make sense?
  - Don't take in more inputs then you need
  - Don't output more then you want

Comments (5 points)

- Did you include your name and date at the top of the file?
- Do you have comments explaining each functions use cases?

Efficiency (5 points)

- Does your code process in a reasonable amount of time?
- Do you have extraneous loops, functions, or variables that don't have any function in your current code?
  - Delete or comment out old code rather then let crud accumulate.

Additional points may be awarded for particularly elegant solutions to complicated problems that go beyond the scope of the class.

Summary

- To figure out why your program isn’t working, think backwards, trying to figure out how what you are seeing could happen.

- Don’t be afraid to look at the documentation.

- “Style” is crucial: good style is important for *you* because it makes it easier to debug programs.