Carl Kingsford, 02-201, Fall 2015

Lecture 8: Style

The goals with style are **Readability & Consistency**.

It's important because it's likely you or someone else will have to modify or maintain this code later.

Design tips

- Break your program down into a collection of small functions, each of which does one thing well. Not just about "style", but small functions let you think about one thing at a time.

Here's a BAD way to write `printDiamond` from last lecture

```go
// BAD CODE BAD CODE BAD CODE
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()
        }
    }
}
// BAD CODE BAD CODE
```

BUG: and in fact the code above has a BUG!

With long functions, you have to understand the entire function all at once.

- Try to design your functions so that you can re-used them for multiple task.
- Don't take in more inputs then you need to a function.
- Don't output more information than you want.

Formatting tips

Indenting to show code structure

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

```
// GOOD:
// AbsInt() computes the absolute value of an integer.
func AbsInt(x int) int {
    if x < 0 {
        return -x
    }
    return x
}

// BAD:
func AbsInt(x int) int {
if x < 0 {return -x
}
return x}
```

Use consistent spacing:

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

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

The `go fmt` command

- 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)

Documentation tips

Use comments appropriately

Go supports two kinds of comments that let you put text in your file that Go will ignore. The first starts with `//` and continues to the end of the line:

```
var x int = 100   // the number of rabbits to start with
```

You can create multi-line comments by starting them with `/*` and ending them with `*/`. For example:

```
/* this function computes the max of 3 numbers.
it will print an error if there is no unique maximum.
*/
```

Each function you write should be accompanied by a short comment that describes what the function does, what data the function expects as input, and what it will return.

Example:

```
// 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
}
```

- Add comments to complicated code to help you remember what it does
- Don't comment obvious code
- Watch out for comments that go out of date: they can mislead you about what your code does

Choose good variable names

If you choose good function and variable names, many comments can be ommited or shortened. They should be descriptive.

```
numSteps, numberOfSteps, nSteps // GOOD names
n // probably a BAD name
```

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).

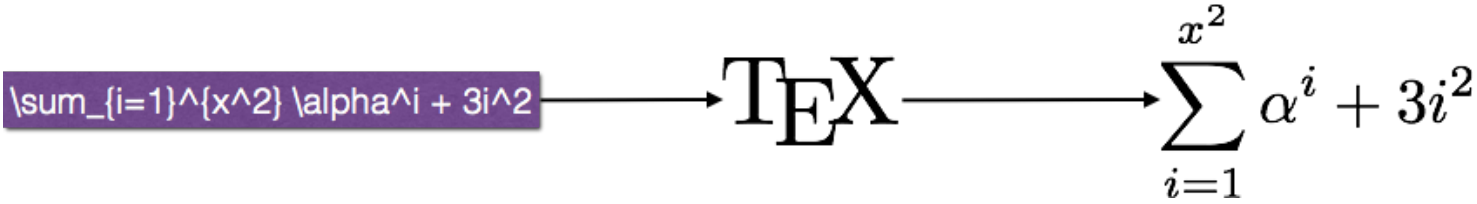
Don’t use the same name for two things. (e.g. don’t use a variable named `KthDigit` inside a function named `KthDigit`)

Debugging tips

Bugs are "normal"

Bugs are not *good*, per se, but it is very hard to write software that is completely 100% bug free.

Example: TeX is a system for typesetting mathematical and scientific papers. It’s was written by Don Knuth (Stanford CS prof) which is very widely used, especially in the sciences.



He started developing it in the 1970s, and has kept track of every bug that he has found and fixed:

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.*

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

Working backwards from the bug

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:

1. I see A, what could have caused A?
2. It could have been B or C or D or E or F.
3. It isn't B because if I comment out that code, I still get the problem.
4. It isn't C because if I add an `if` statement to check if C is happening, I see that it is not.
5. It isn't D because I wrote a small test program, and D can't happen.
6. It isn't E because I print out the value of E, and it's correct.
7. So it must be F.

Then repeat.

Use print statements

1. Add `fmt.Println()` statements at various points to check the value of variables.
2. Add `fmt.Println()` statements to check which lines of code are actually being executed.

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.