UCI Division of Continuing Education

Module 1: Functions and Organization
Topic 1.1: Why Use Functions?

What is a Function

- A set of instructions with a name (usually)

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

```
func PrintHello() {
    fmt.Printf("Hello, world.")
}
func main() {
    PrintHello()
}
```

- Function declaration, name, call

Reusability

- You only need to declare a function once
- Good for commonly used operations
- Graphics editing program might have `ThresholdImage()`
- Database program might have `QueryDbase()`
- Music program might have `ChangeKey()`

Abstraction

- Details are hidden in the function
- Only need to understand input/output behavior
- Improves understandability

```
func FindPupil() {
   GrabImage()
   FilterImage()
   FindEllipses()
}
```

- Naming is important for clarity

Module 1: Functions and Organization
Topic 1.2: Function Parameters and Return values

Function Parameters

- Functions may need input data to perform their operations
- **Parameters** are listed in parenthesis after function name
- **Arguments** are supplied in the call

```
func foo(x int, y int) {
   fmt.Print(x * y)
}
func main() {
   foo(2, 3)
}
```

Parameter Options

- If no parameters are needed, put nothing in parentheses
- Still need parentheses

```
func foo() {
}
```

- List arguments of same type

```
func foo(x, y int) {
}
```

Return Values

- Functions can return a value as a result
- **Type of return value** after parameters in declaration
- Function call used on right-hand side of an assignment

```
func foo(x int) int {
   return x + 1
}
y := foo(1)
```

Multiple Return Values

- Multiple value types must be listed in the declaration

```
func foo2(x int) (int, int) {
    return x, x + 1
}
a, b := foo2(3)
```

Module 1: Functions and Organization
Topic 1.3: Call by Value, Reference

Call by Value

- Passed arguments are copied to parameters
- Modifying parameters has no effect outside the function

```
func foo(y int) {
   y = y + 1
}
func main() {
   x := 2
   foo(x)
   fmt.Print(x)
}
```

UCI Division of Continuing Education

Tradeoffs of Call by Value

- **Advantage: Data Encapsulation**
- Function variables only changed inside the function
- **Disadvantage: Copying Time**
- Large objects may take a long time to copy

Call by Reference

- Programmer can **pass a pointer** as an argument
- Called function has direct access to caller variable in memory

```
func foo(y *int) {
    *y = *y + 1
}
func main() {
    x := 2
    foo(&x)
    fmt.Print(x)
}
```

Tradeoffs of Call by Reference

- **Advantage: Copying Time**
- Don’t need to copy arguments
- **Disadvantage: Data Encapsulation**
- Function variables may be changed in called functions
- May be what you want
  - Sort an array

Module 1: Functions and Organization
Topic 1.4: Passing Arrays and Slices

Passing Array Arguments

- Array arguments are copied
- Arrays can be big, so this can be a problem

```
func foo(x [3]int) int {
   return x[0]
}
func main() {
   a := [3]int{1, 2, 3}
   fmt.Print(foo(a))
}
```

Passing Array Pointers

- Possible to pass array pointers

```
func foo(x *[3]int) {
    (*x)[0] = (*x)[0] + 1
}
func main() {
    a := [3]int{1, 2, 3}
    foo(&a)
    fmt.Print(a)
}
```

- Messy and unnecessary

Pass Slices Instead

- **Slices contain a pointer** to the array
- Passing a slice copies the pointer

```
func foo(sli []int) {
    sli[0] = sli[0] + 1
}
func main() {
    a := []int{1, 2, 3}
    foo(a)
    fmt.Print(a)
}
```

Module 1: Functions and Organization
Topic 2.1: Well-written Functions

Understandability

- Code is **functions** and **data**
- If you are asked to **find a feature**, you can find it quickly
  - “Where is the function that blurs the image?”
  - “Where do you compute the average score?”
- If you are asked about **where data is used**, you know
  - “Where do you modify the record list?”
  - “Where do you access the file?”

Debugging Principles

- Code crashes inside a function
- Two options for the cause

1. **Function is written incorrectly**
   - Sorts a slice in the wrong order
2. **Data that the function uses is incorrect**
   - Sorts slice correctly but slice has wrong elements in it

Supporting Debugging

- Functions need to be understandable
  - Determine if actual behavior matches desired behavior
- Data needs to be traceable
  - Where did the input data come from?
  - Global variables complicate this

Module 1: Functions and Organization
Topic 2.2: Guidelines for Functions

Function Naming

- Give functions a good name
  - Behavior can be understood at a glance
  - Parameter naming counts too

```
func ProcessArray (a []int)
   float {}
func ComputeRMS (samples
[]float) float {}
```

- RMS = Root Mean Square
- `samples` is a slice of samples of a time-varying signal

Functional Cohesion

- Function should perform **only one "operation"**
- An "operation" depends on the context
- Example: Geometry application
- Good functions:
  - `PointDist(), DrawCircle(), TriangleArea()`
- Merging behaviors makes code complicated
  - `DrawCircle() + TriangleArea()`

Few Parameters

- Debugging requires tracing function input data
- More difficult with a large number of parameters
- Function may have bad functional cohesion
  - `DrawCircle()` and `TriangleArea()` require different arguments

Reducing Parameter Number

- May need to group related arguments into structures
- `TriangleArea()`, bad solution
  - 3 points needed to define triangle
  - Each point has 3 floats (in 3D)
  - Total, 9 arguments
- `TriangleArea()`, good solution

```
type Point struct{x, y, z float}
```

  - Total, 3 arguments

Module 1: Functions and Organization
Topic 2.3: Function Guidelines

Function Complexity

- Functions should be simple
  - Easier to debug
- **Function length** is the most obvious measure
- Short functions can be complicated too

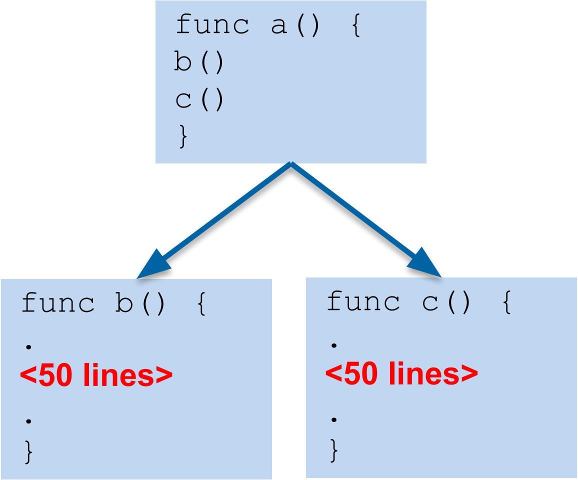
Function Length

- How do you write complicated code with simple functions?
- **Function Call Hierarchy**

Option 1

```
func a() {
.
<100 lines>
.
}
```

Option 2



```
func a() {
b()
c()
}
```

```
func b() {
.
<50 lines>
.
}
```

```
func c() {
.
<50 lines>
.
}
```

Control-flow Complexity

- Control-flow describes conditional paths

```
func foo() {
    if a == 1 {
        if b == 1 {
            …
        }
    }
    …
}
```

- 3 control-flow paths

Partitioning Conditionals

- Functional hierarchy can reduce control-flow complexity

```
func foo() {
    if a == 1 {
        CheckB()
    }
    …
}
```

```
func CheckB() {
    if b == 1 {
        …
    }
}
```

- 2 control-flow paths in each function