EECS 22: Advanced C Programming

Lecture 7

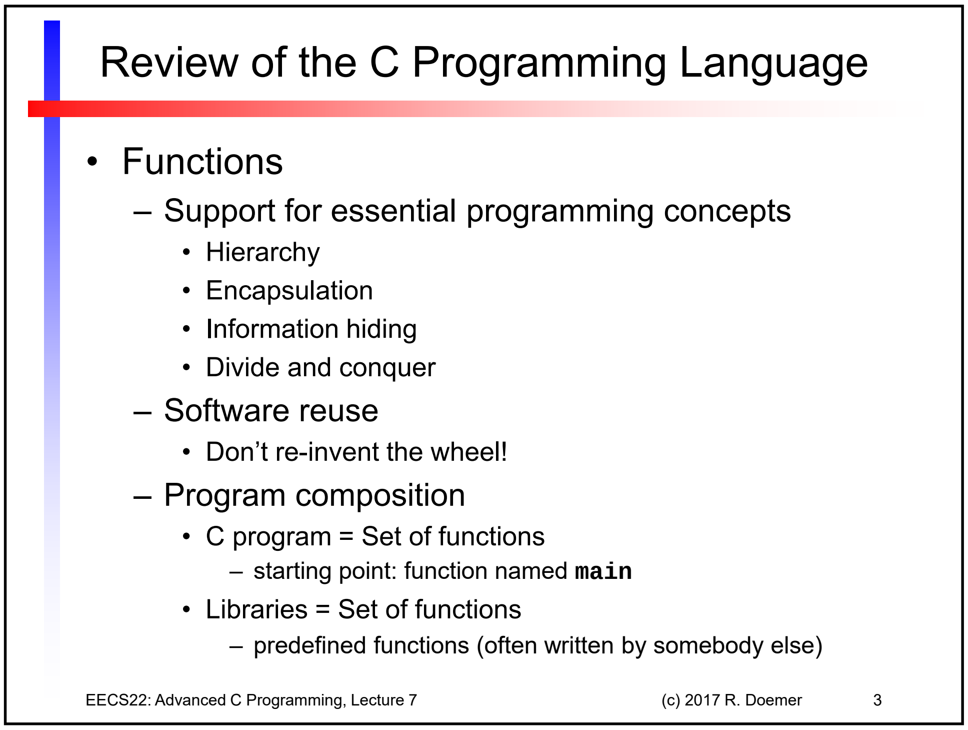
Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 7: Overview

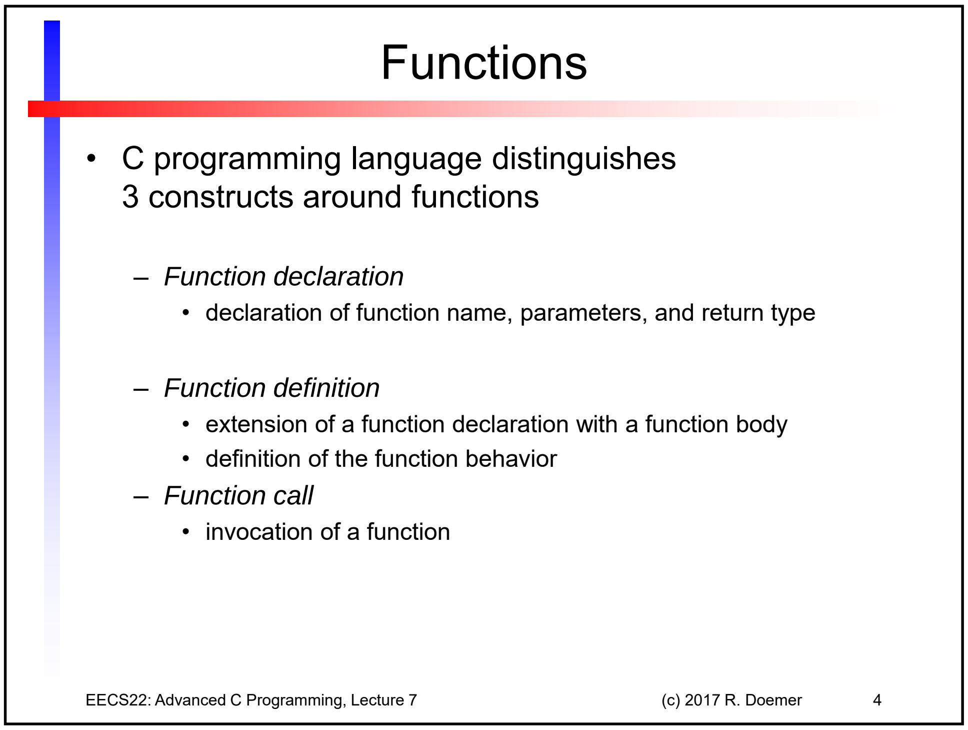
- Review of the C Programming Language
  - Functions
    - Introduction and concept
    - Declaration, definition, and function call
  - Passing Arguments to Function Parameters
    - Pass by value vs. pass by reference
- Hierarchy of Functions
  - Example program **`Cylinder.c`**
  - Function call graph
  - Function call trace
  - Function call stack
  - Long Jump



Review of the C Programming Language

- Functions
  - Support for essential programming concepts
    - Hierarchy
    - Encapsulation
    - Information hiding
    - Divide and conquer
  - Software reuse
    - Don’t re-invent the wheel!
  - Program composition
    - C program = Set of functions
      - starting point: function named **`main`**
    - Libraries = Set of functions
      - predefined functions (often written by somebody else)

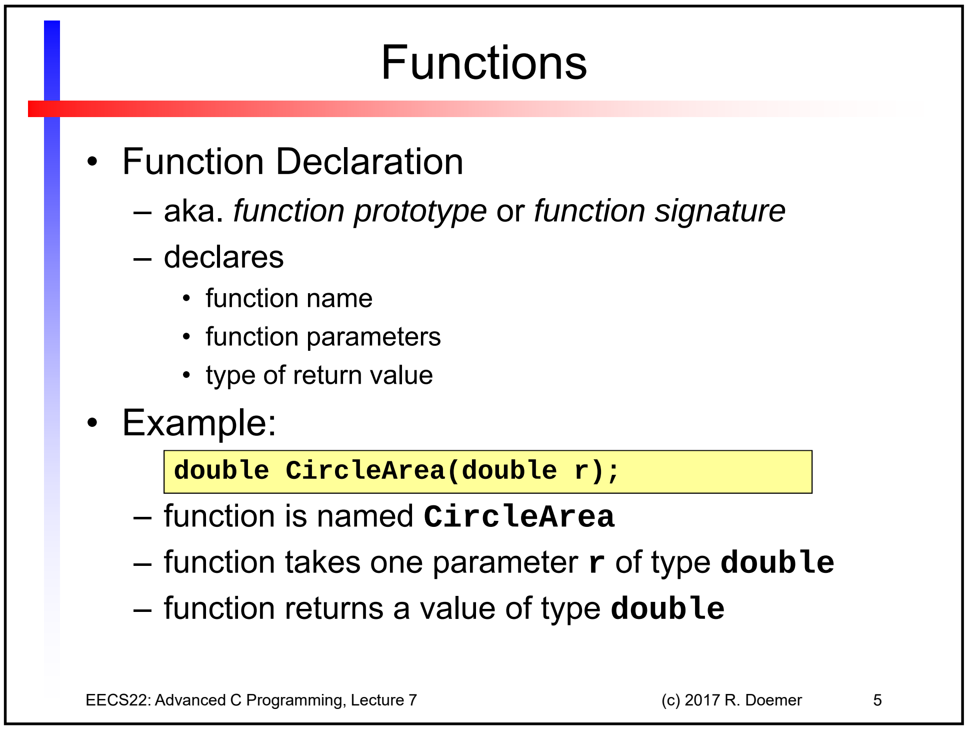
EECS22: Advanced C Programming, Lecture 7 (c) 2017 R. Doemer 3



Functions

- C programming language distinguishes 3 constructs around functions
  - *Function declaration*
    - declaration of function name, parameters, and return type
  - *Function definition*
    - extension of a function declaration with a function body
    - definition of the function behavior
  - *Function call*
    - invocation of a function

EECS22: Advanced C Programming, Lecture 7 (c) 2017 R. Doemer 4

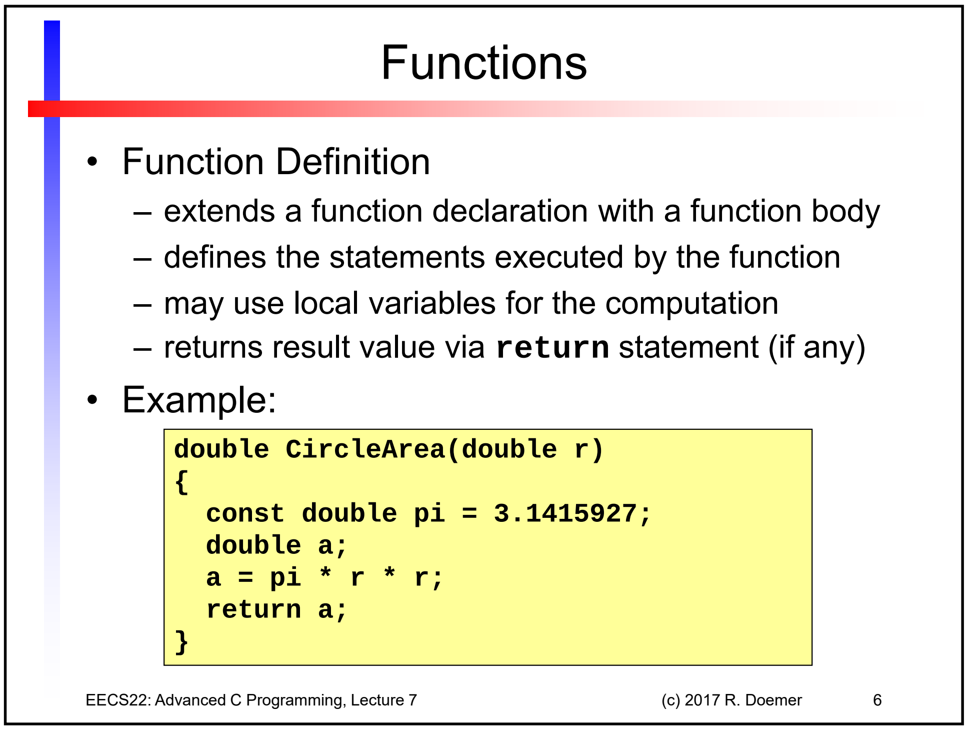


Functions

- Function Declaration
  - aka. *function prototype* or *function signature*
  - declares
    - function name
    - function parameters
    - type of return value
- Example:

```
double CircleArea(double r);
```

  - function is named **`CircleArea`**
  - function takes one parameter **`r`** of type **`double`**
  - function returns a value of type **`double`**



Functions

- Function Definition
  - extends a function declaration with a function body
  - defines the statements executed by the function
  - may use local variables for the computation
  - returns result value via **`return`** statement (if any)
- Example:

```
double CircleArea(double r)
{
  const double pi = 3.1415927;
  double a;
  a = pi * r * r;
  return a;
}
```

Functions

- Function Call
  - expression invoking a function
  - supplies arguments for formal parameters
  - invokes the function
  - result is the value returned by the function
- Example:

```
double a, b = 10.0;
a = CircleArea(b);
```

  - function **`CircleArea`** is called
  - argument **`b`** is passed for parameter **`r`** (by value)
  - value returned by the function is assigned to **`a`**

Functions

- C Programming Language distinguishes 3 Constructs
  - Function declaration
    - declaration of function name, parameters, and return type
  - Function definition
    - extension of a function declaration with a function body
    - definition of the function behavior
  - Function call
    - invocation of a function
- C Program Rules
  - A function must be declared before it can be called.
  - Multiple function declarations are allowed (if they match).
  - A function definition is an implicit function declaration.
  - A function must be defined exactly once in a program.
  - A function may be called any number of times.

Passing Arguments to Functions

- In ANSI C, ...
  - ... basic types are passed by value
  - ... arrays are passed by reference
- Pass by Value
  - only the *current value* is passed as argument
  - the parameter is a *copy* of the argument
  - changes to the parameter *do not* affect the argument
- Pass by Reference
  - a *reference* to the object is passed as argument
  - the parameter is a *reference* to the argument
  - changes to the parameter *do* affect the argument

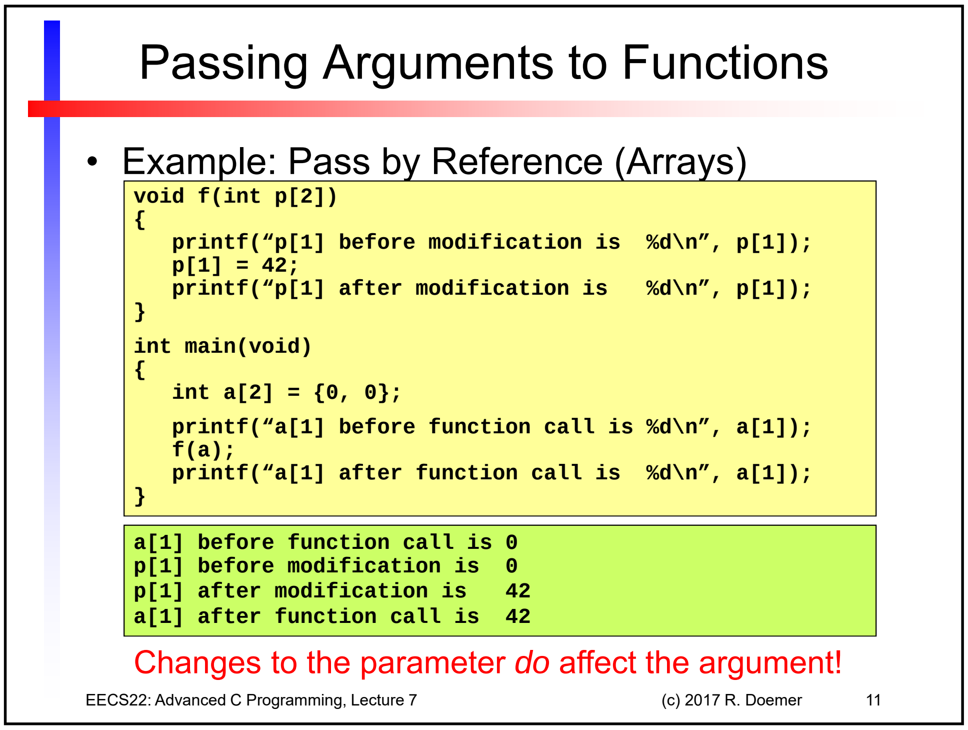
Passing Arguments to Functions

- Example: Pass by Value (Basic Types)

```
void f(int p)
{
   printf(“p before modification is  %d\n”, p);
   p = 42;
   printf(“p after modification is   %d\n”, p);
}
int main(void)
{
   int a = 0;
   printf(“a before function call is %d\n”, a);
   f(a);
   printf(“a after function call is  %d\n”, a);
}
```

```
a before function call is 0
p before modification is  0
p after modification is   42
a after function call is  0
```

Changes to the parameter *do not* affect the argument!



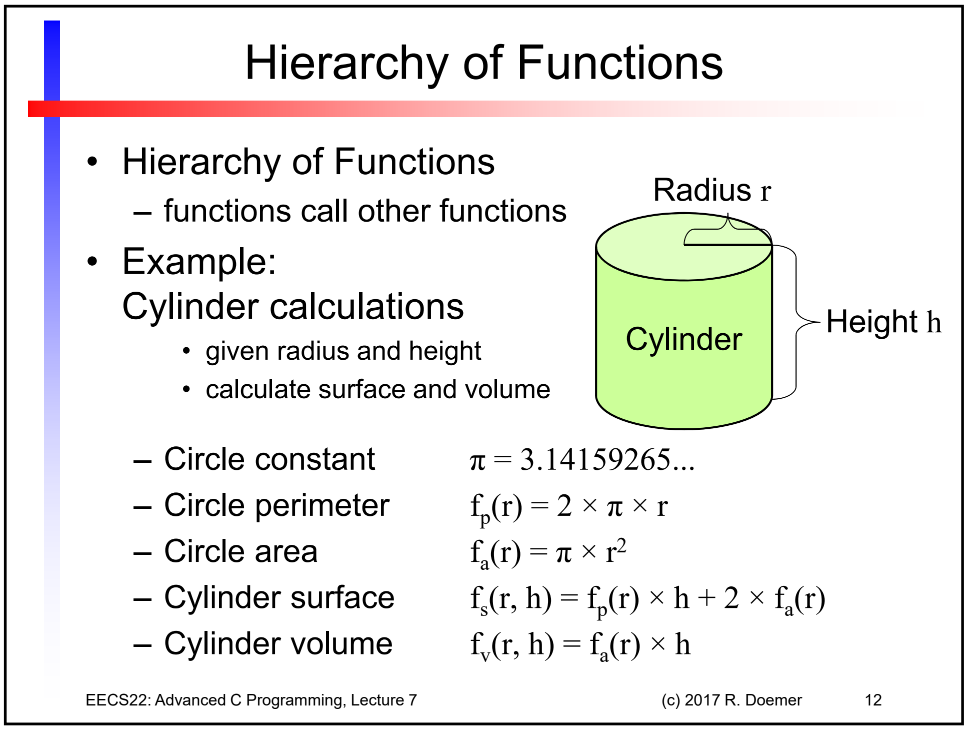
Passing Arguments to Functions

- Example: Pass by Reference (Arrays)

```
void f(int p[2])
{
   printf("p[1] before modification is  %d\n", p[1]);
   p[1] = 42;
   printf("p[1] after modification is   %d\n", p[1]);
}
int main(void)
{
   int a[2] = {0, 0};
   printf("a[1] before function call is %d\n", a[1]);
   f(a);
   printf("a[1] after function call is  %d\n", a[1]);
}
```

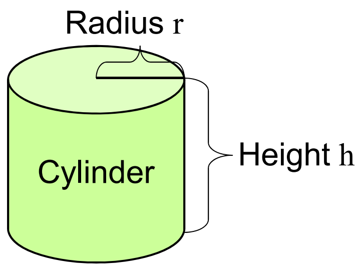
```
a[1] before function call is 0
p[1] before modification is  0
p[1] after modification is   42
a[1] after function call is  42
```

Changes to the parameter *do* affect the argument!

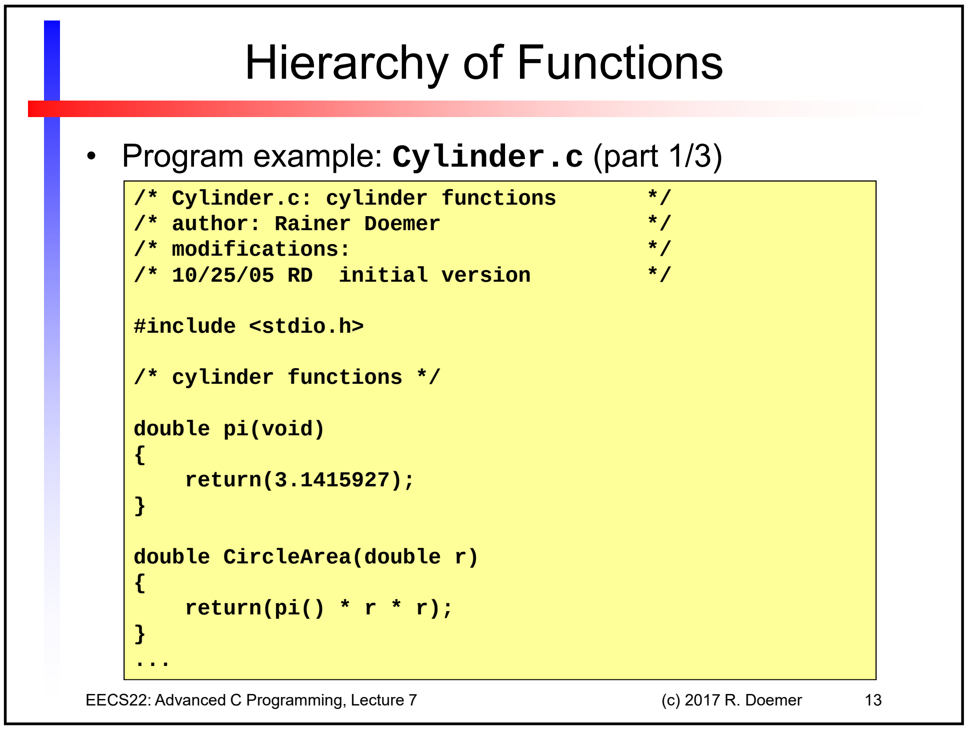


Hierarchy of Functions

- Hierarchy of Functions
  - functions call other functions
- Example:
  Cylinder calculations
  - given radius and height
  - calculate surface and volume



– Circle constant	$\pi = 3.14159265...$
– Circle perimeter	$f_p(r) = 2 \times \pi \times r$
– Circle area	$f_a(r) = \pi \times r^2$
– Cylinder surface	$f_s(r, h) = f_p(r) \times h + 2 \times f_a(r)$
– Cylinder volume	$f_v(r, h) = f_a(r) \times h$

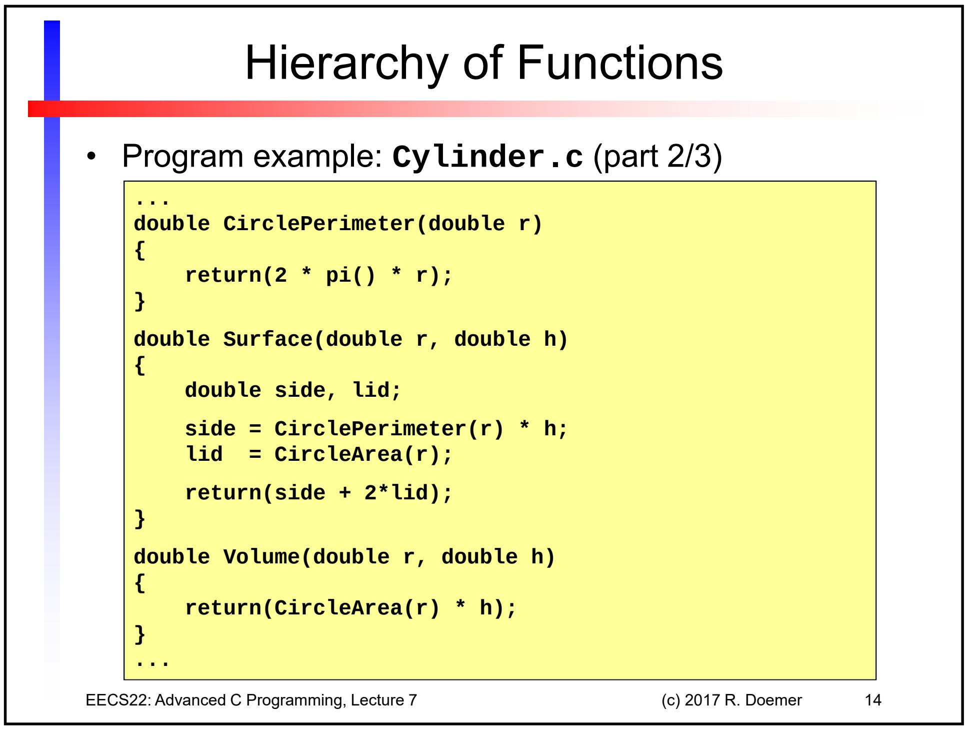


Hierarchy of Functions

- Program example: **`Cylinder.c`** (part 1/3)

```
/* Cylinder.c: cylinder functions       */
/* author: Rainer Doemer                */
/* modifications:                       */
/* 10/25/05 RD  initial version         */

#include <stdio.h>

/* cylinder functions */

double pi(void)
{
    return(3.1415927);
}

double CircleArea(double r)
{
    return(pi() * r * r);
}
...
```

EECS22: Advanced C Programming, Lecture 7 (c) 2017 R. Doemer 13

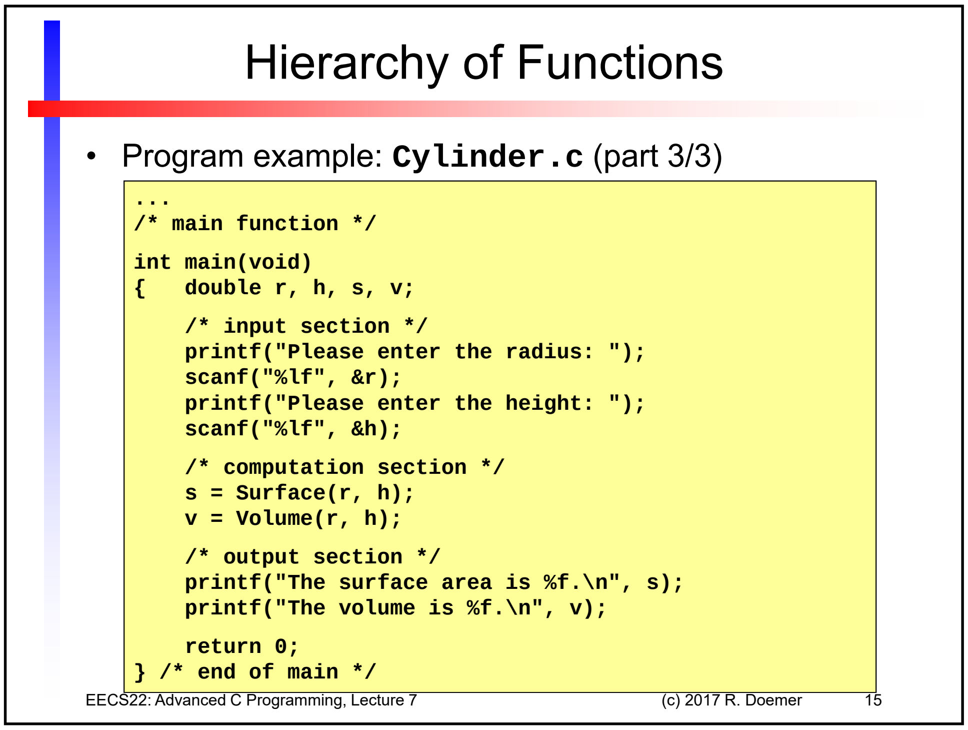


Hierarchy of Functions

- Program example: **`Cylinder.c`** (part 2/3)

```
...
double CirclePerimeter(double r)
{
    return(2 * pi() * r);
}
double Surface(double r, double h)
{
    double side, lid;
    side = CirclePerimeter(r) * h;
    lid  = CircleArea(r);
    return(side + 2*lid);
}
double Volume(double r, double h)
{
    return(CircleArea(r) * h);
}
...
```

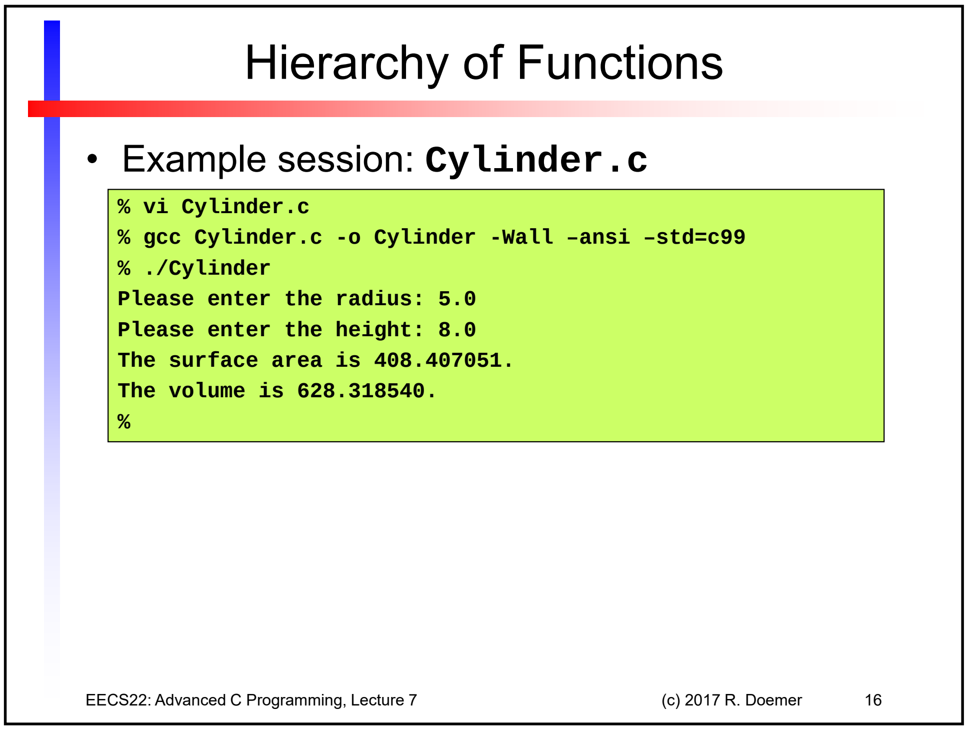
EECS22: Advanced C Programming, Lecture 7 (c) 2017 R. Doemer 14



Hierarchy of Functions

- Program example: **`Cylinder.c`** (part 3/3)

```
...
/* main function */

int main(void)
{   double r, h, s, v;

    /* input section */
    printf("Please enter the radius: ");
    scanf("%lf", &r);
    printf("Please enter the height: ");
    scanf("%lf", &h);

    /* computation section */
    s = Surface(r, h);
    v = Volume(r, h);

    /* output section */
    printf("The surface area is %f.\n", s);
    printf("The volume is %f.\n", v);

    return 0;
} /* end of main */
```



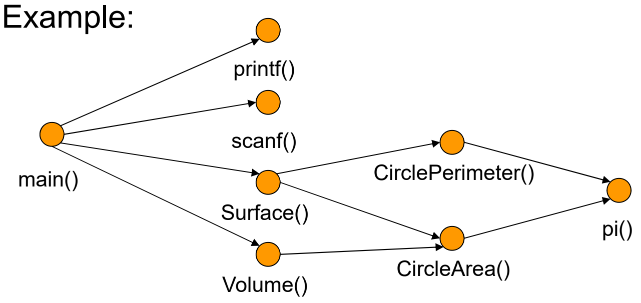
Hierarchy of Functions

- Example session: **`Cylinder.c`**

```
% vi Cylinder.c
% gcc Cylinder.c -o Cylinder -Wall –ansi –std=c99
% ./Cylinder
Please enter the radius: 5.0
Please enter the height: 8.0
The surface area is 408.407051.
The volume is 628.318540.
%
```

Function Call Graph

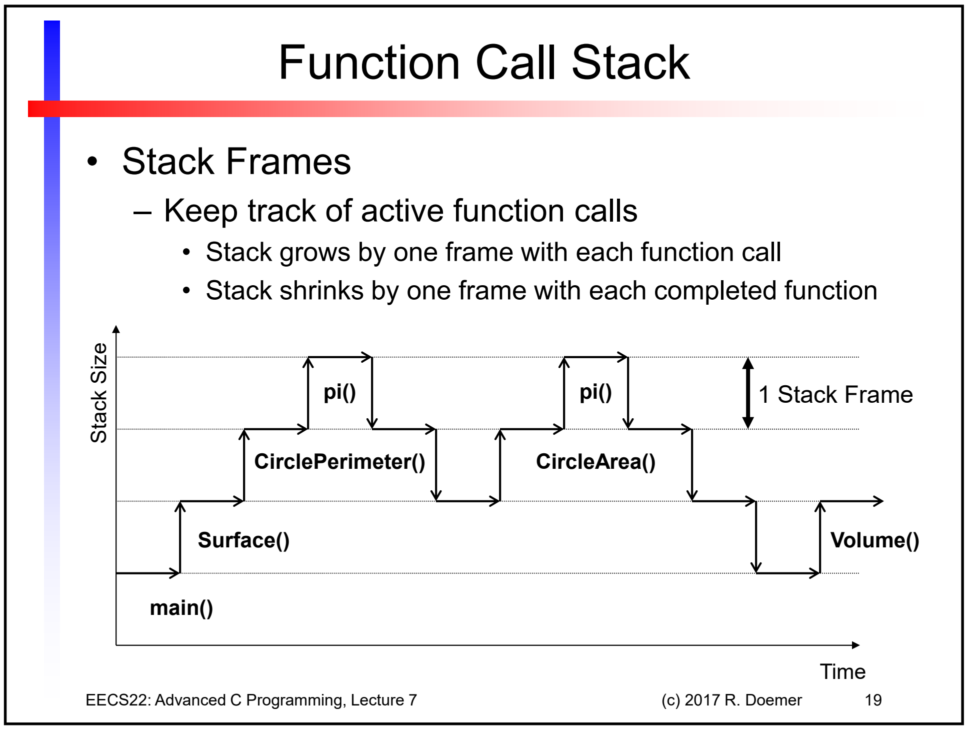
- Graphical Representation of Function Calls
  - Directed Graph
    - Nodes: Functions
    - Edges: Function calls
  - Shows dependencies among functions
  - Example:



Function Call Trace

- Sequence of Function Calls
  - shows execution order of functions at run-time
- Example:

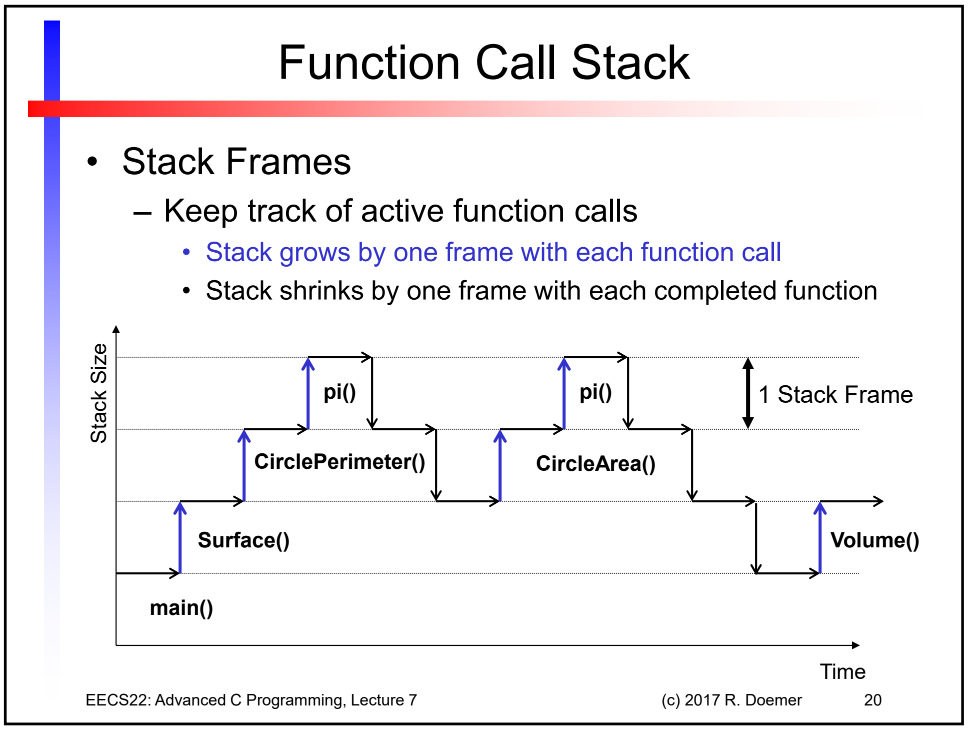
```
– main()
    • printf()
    • scanf()
    • printf()
    • scanf()
    • Surface()
        – CirclePerimeter()
            » pi()
        – CircleArea()
            » pi()
    • Volume()
        – CircleArea()
            » pi()
    • printf()
    • printf()
```



Function Call Stack

- Stack Frames
  - Keep track of active function calls
    - Stack grows by one frame with each function call
    - Stack shrinks by one frame with each completed function

Stack Size
main()
Surface()
CirclePerimeter()
pi()
CircleArea()
pi()
Volume()
1 Stack Frame
Time



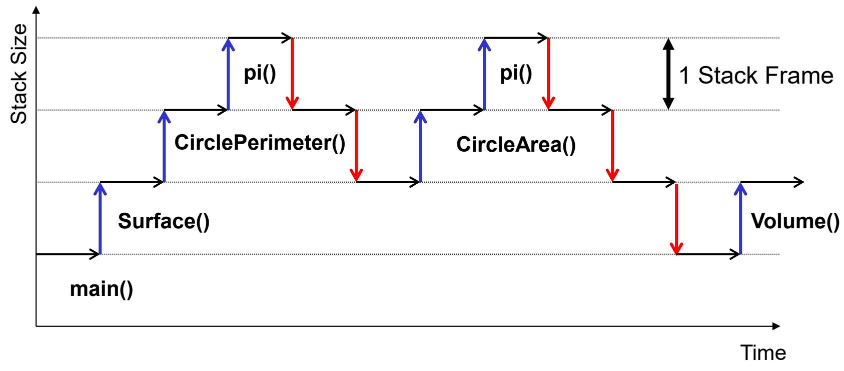
Function Call Stack

- Stack Frames
  - Keep track of active function calls
    - Stack grows by one frame with each function call
    - Stack shrinks by one frame with each completed function

Stack Size
main()
Surface()
CirclePerimeter()
pi()
CircleArea()
pi()
Volume()
1 Stack Frame
Time

Function Call Stack

- Stack Frames
  - Keep track of active function calls
    - Stack grows by one frame with each function call
    - Stack shrinks by one frame with each completed function



Non-Local Goto: Long Jump

- *Long Jump*: Returning to a previous stack frame
  - Useful, for example, when dealing with errors (or interrupts) in a low-level function of a program.
    - ➢ However, long jumps are hard to understand and maintain!
    - ➢ Same as goto, avoid long jumps, if possible!

  -
    ```
    #include <setjmp.h>
    int setjmp(jmp_buf env);
    ```
    - saves current stack context in **env** for later use by **`longjmp()`**
    - stack context in **env** is valid until the function which called **`setjmp()`** returns
  - `void longjmp(jmp_buf env, int val);`
    - non-local jump (return) to a saved stack context **env**
    - **`longjmp()`** restores the stack context saved by **`setjmp()`**
    - after **`longjmp()`**, program execution continues as if the call of **`setjmp()`** had just returned the value **`val`**

Non-Local Goto: Long Jump

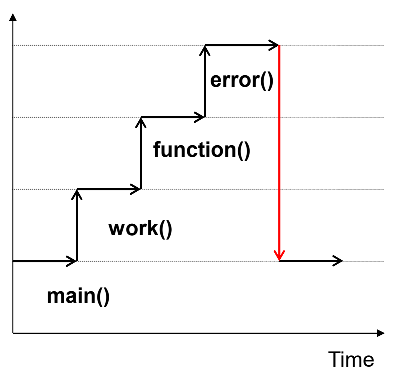
- *Long Jump*: Returning to a previous stack frame
- Example:

```
#include <setjmp.h>
jmp_buf env;        /* storage for stack context */
void error(void)    /* error, return to main! */
{
  longjmp(env, 1);
}
int main(void)
{
  if (setjmp(env))  /* store current stack context */
  { /* long jump arrives here! */
    return 10;
  }
  work(...); /* call tree can call error at any time */
  return 0;
}
```

Non-Local Goto: Long Jump

- *Long Jump*: Returning to a previous stack frame
- Example:

```
#include <setjmp.h>
jmp_buf env;
void error(void)
{
  longjmp(env, 1);
}
int main(void)
{
  if (setjmp(env))
  { // long jump target
    return 10;
  }
  work(...);
  return 0;
}
```


error()
function()
work()
main()
Time