ICS103: Programming in C

3: Top-Down Design with Functions

Prof. Muhamed F. Mudawar

OUTLINE

- **Building Programs from Existing Information**
- **Library Functions and Code Reuse**
- Top-Down Design and Structure Charts
- Functions, Prototypes, and Definitions
- Functions with Arguments
- Testing Functions and Function Data Area
- Advantages of Functions and Common Errors

Recall: Software Development Method

1. **Specify** the problem
2. **Analyze** the problem
3. **Design** the algorithm to solve the problem
4. **Implement** the algorithm
5. **Test** and verify the completed program
6. **Maintain** and update the program

Case Study: Computing the Weight of a Batch of Flat Washers

- **1. Problem:** Write a program that computes the weight of a specified quantity of flat washers.
- **2. Analysis:** to compute the weight of a single flat washer, you should know its area, thickness, and density.

  Inputs:

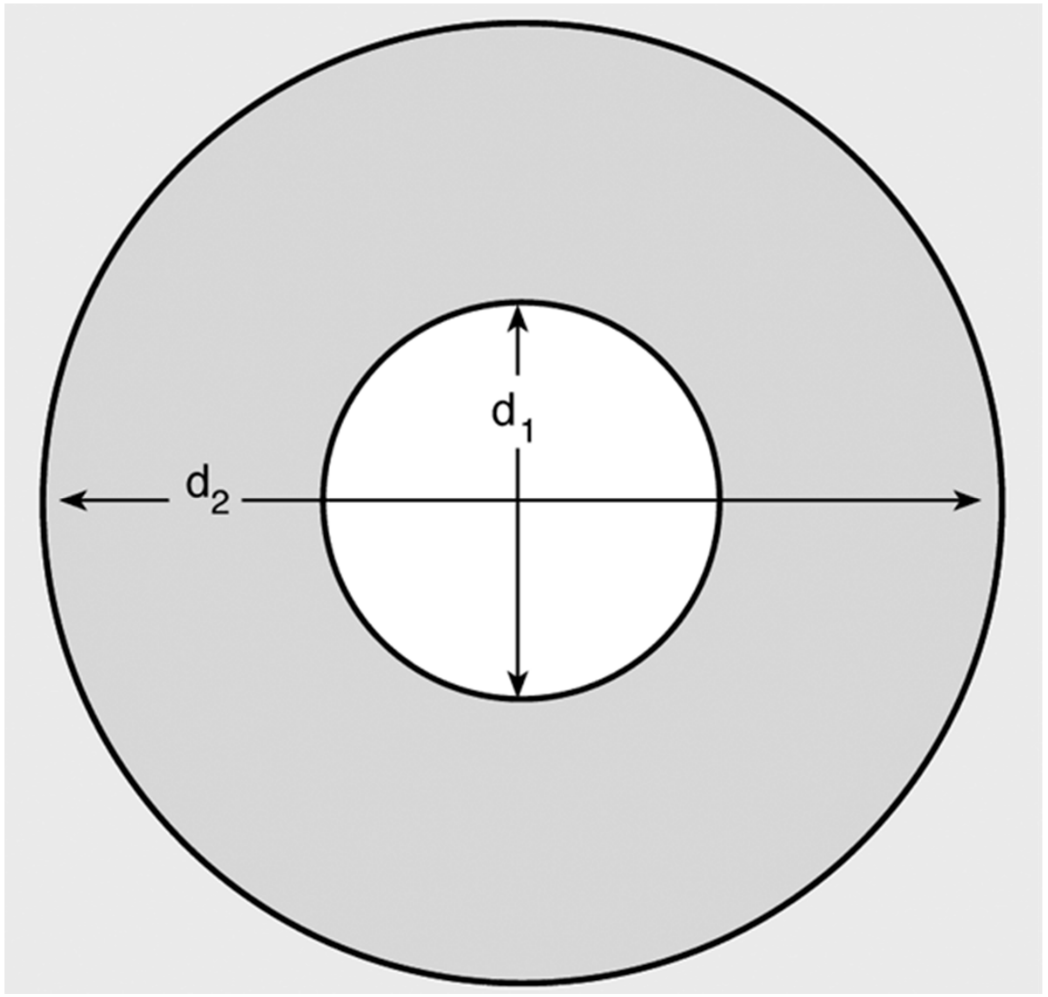
  hole diameter, edge diameter, thickness, density, quantity

  Output:

  weight (of a batch of flat washers)

2. Analysis

Computing the Area and Weight



rim area =

$$\pi(d_2/2)^2 - \pi(d_1/2)^2$$

unit weight =

rim area $\times$

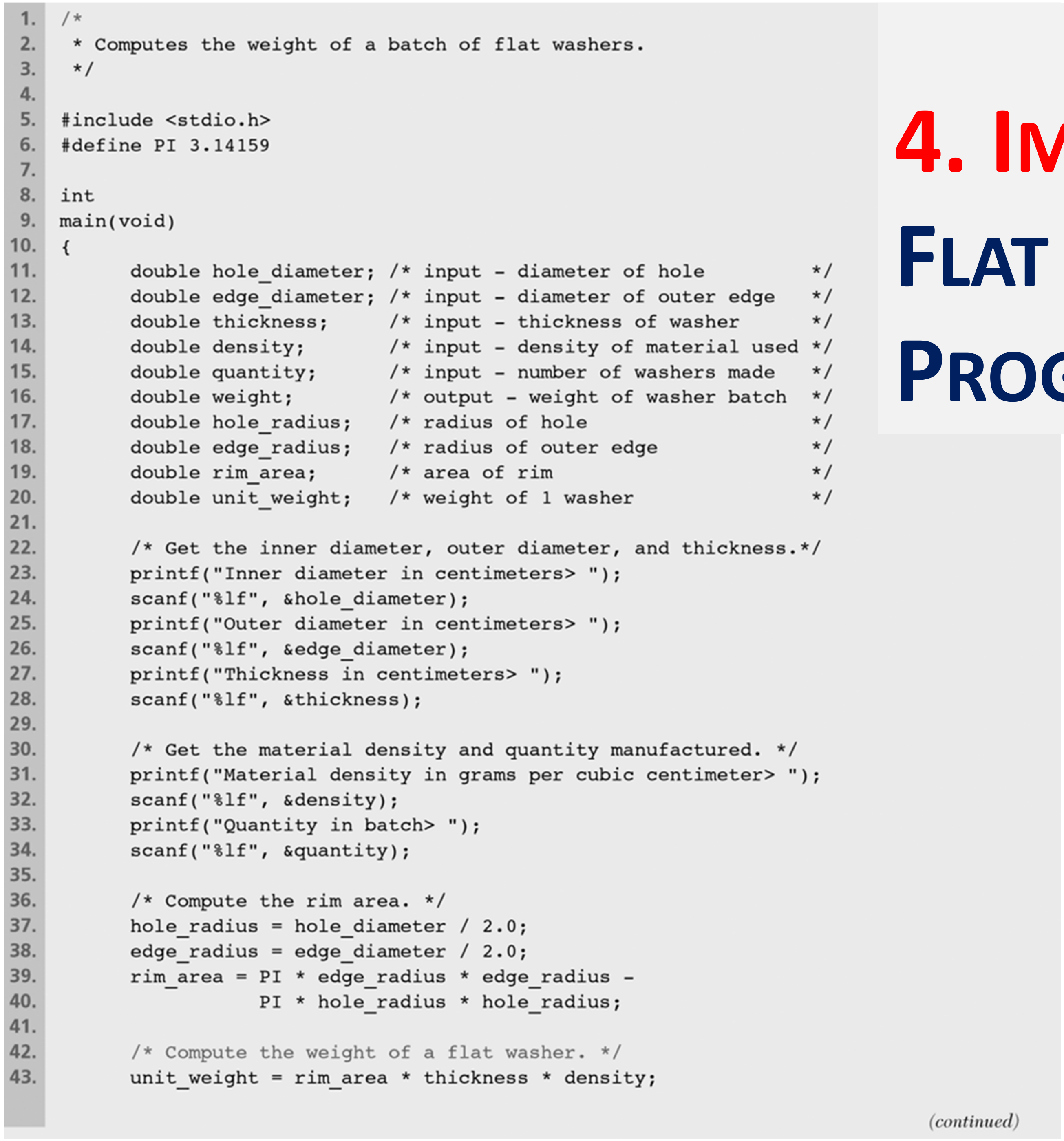
thickness $\times$

density

3. Designing The Algorithm

1. Read the washer's inner diameter, outer diameter, and thickness
2. Read the material density and quantify of washers
3. Compute the rim area
4. Compute the weight of one flat washer
5. Compute the weight of the batch of washers
6. Display the weight of the batch of washers

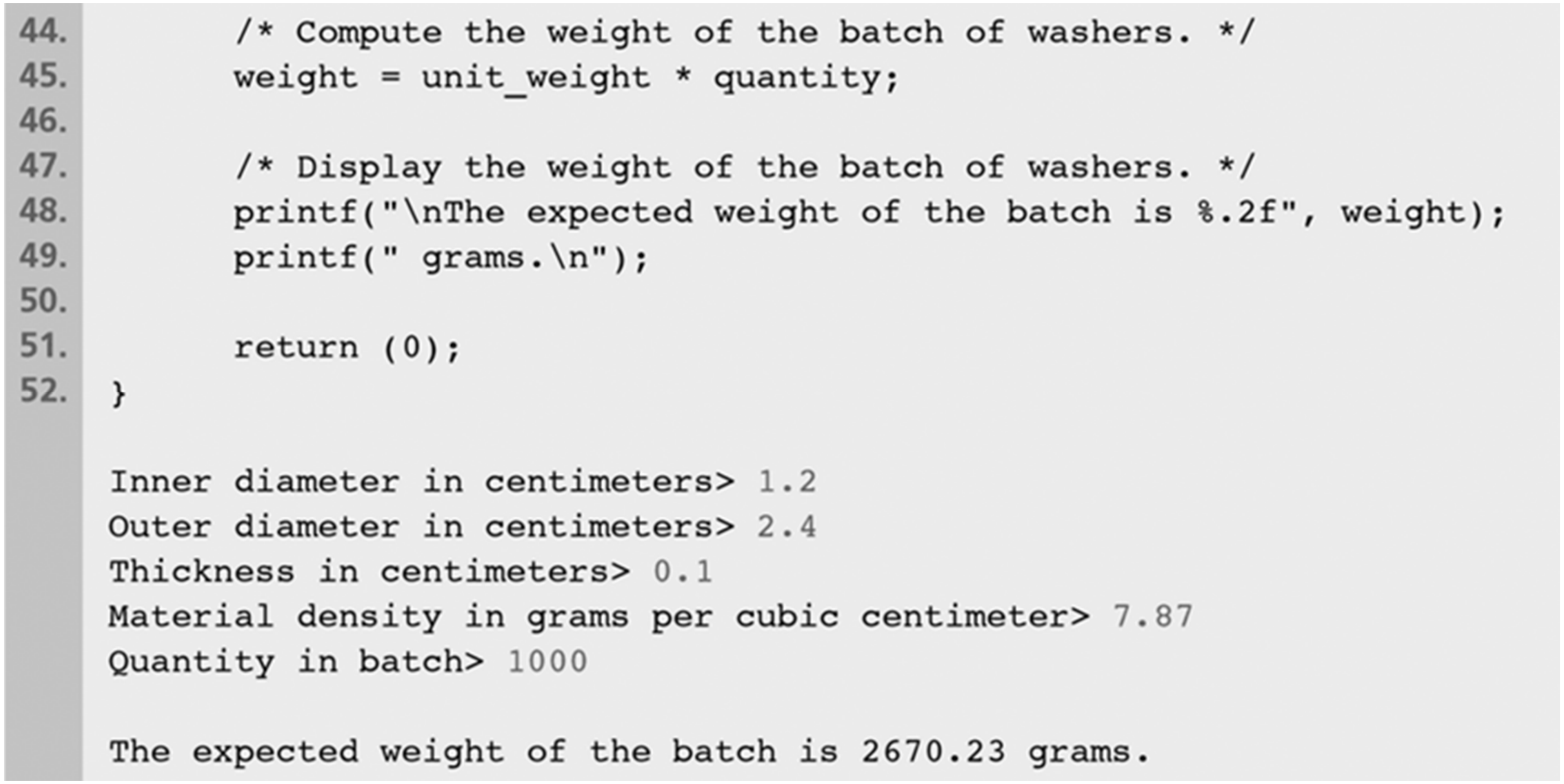
4. Implement Flat Washer Program



```c
/*
 * Computes the weight of a batch of flat washers.
 */

#include <stdio.h>
#define PI 3.14159

int
main(void)
{
      double hole_diameter; /* input - diameter of hole          */
      double edge_diameter; /* input - diameter of outer edge    */
      double thickness;     /* input - thickness of washer       */
      double density;       /* input - density of material used */
      double quantity;      /* input - number of washers made    */
      double weight;        /* output - weight of washer batch   */
      double hole_radius;   /* radius of hole                    */
      double edge_radius;   /* radius of outer edge              */
      double rim_area;      /* area of rim                       */
      double unit_weight;   /* weight of 1 washer                */

      /* Get the inner diameter, outer diameter, and thickness.*/
      printf("Inner diameter in centimeters> ");
      scanf("%lf", &hole_diameter);
      printf("Outer diameter in centimeters> ");
      scanf("%lf", &edge_diameter);
      printf("Thickness in centimeters> ");
      scanf("%lf", &thickness);

      /* Get the material density and quantity manufactured. */
      printf("Material density in grams per cubic centimeter> ");
      scanf("%lf", &density);
      printf("Quantity in batch> ");
      scanf("%lf", &quantity);

      /* Compute the rim area. */
      hole_radius = hole_diameter / 2.0;
      edge_radius = edge_diameter / 2.0;
      rim_area = PI * edge_radius * edge_radius -
                 PI * hole_radius * hole_radius;

      /* Compute the weight of a flat washer. */
      unit_weight = rim_area * thickness * density;
```

(continued)

Flat Washer Program (Cont'd)



```
    /* Compute the weight of the batch of washers. */
    weight = unit_weight * quantity;

    /* Display the weight of the batch of washers. */
    printf("\nThe expected weight of the batch is %.2f", weight);
    printf(" grams.\n");

    return (0);
}

Inner diameter in centimeters> 1.2
Outer diameter in centimeters> 2.4
Thickness in centimeters> 0.1
Material density in grams per cubic centimeter> 7.87
Quantity in batch> 1000

The expected weight of the batch is 2670.23 grams.
```

5. Testing

Run the program with inner, outer diameters, thickness, and densities that lead to calculations that can be verified easily.

Library Functions and Code Reuse

- The primary goal of software engineering is to write error-free code.
- Reusing code that has already been written and tested is one way to achieve this.
- C promotes code reuse by providing library functions.
  - Input/Output functions: **`printf`**, **`scanf`** , etc.
  - Mathematical functions: **`sqrt`**, **`exp`**, **`log`**, etc.
  - String functions: **`strlen`**, **`strcpy`**, **`strcmp`**, etc.
- Appendix B lists many C standard library functions

Some Mathematical Library Functions

Function	Header file	Argument	Result	Example
`abs(x)`	`<stdlib.h>`	`int`	`int`	`abs(-5) is 5`
`fabs(x)`	`<math.h>`	`double`	`double`	`fabs(-2.3) is 2.3`
`sqrt(x)`	`<math.h>`	`double`	`double`	`sqrt(2.25) is 1.5`
`exp(x)`	`<math.h>`	`double`	`double`	`exp(1.0) is 2.71828`
`log(x)`	`<math.h>`	`double`	`double`	`log(2.71828) is 1.0`
`log10(x)`	`<math.h>`	`double`	`double`	`log10(100.0) is 2.0`
`pow(x,y)`	`<math.h>`	`double, double`	`double`	`pow(2.0,3.0) is 8.0` returns x^y
`sin(x)`	`<math.h>`	`double`	`double`	`sin(PI/2.0) is 1.0`
`cos(x)`	`<math.h>`	`double`	`double`	`cos(PI/3.0) is 0.5`
`tan(x)`	`<math.h>`	`double`	`double`	`tan(PI/4.0) is 1.0`
`ceil(x)`	`<math.h>`	`double`	`double`	`ceil(45.2) is 46.0`
`floor(x)`	`<math.h>`	`double`	`double`	`floor(45.2) is 45.0`

USING MATH LIBRARY FUNCTIONS

```
#include <math.h>
```

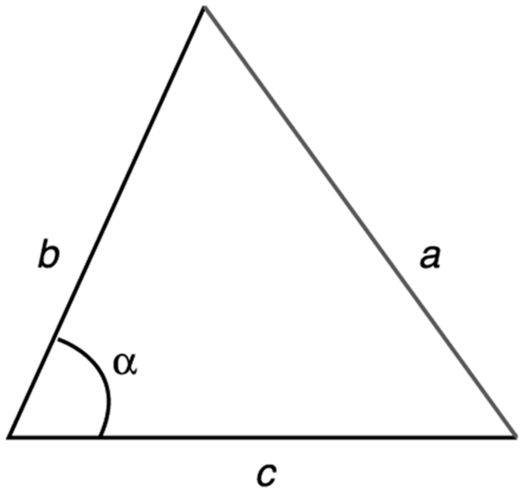
- Computing the roots of: $ax^2 + bx + c = 0$

```
delta = b*b – 4*a*c;
root1 = (-b + sqrt(delta))/(2.0 * a);
root2 = (-b - sqrt(delta))/(2.0 * a);
```

- Computing the unknown side of a triangle
- $a^2 = b^2 + c^2 - 2\, b\, c \cos(\alpha)$

```
a = sqrt(b*b + c*c -
        2*b*c*cos(alpha));
```

- **`alpha`** must be in radians



NEXT . . .

- Building Programs from Existing Information
- Library Functions and Code Reuse
- **Top-Down Design and Structure Charts**
- **Functions, Prototypes, and Definitions**
- Functions with Arguments
- Testing Functions and Function Data Area
- Advantages of Functions and Common Errors

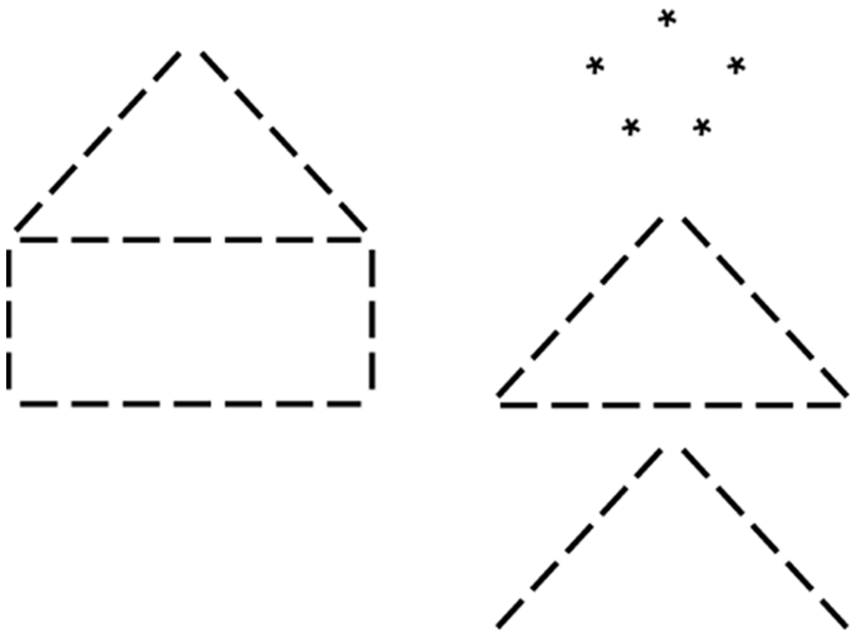
Top-Down Design

- Algorithms are often complex
- To solve a problem, the programmer must break it into sub-problems at a lower level
- This process is called **top-down design**
- **Examples:**

  Drawing

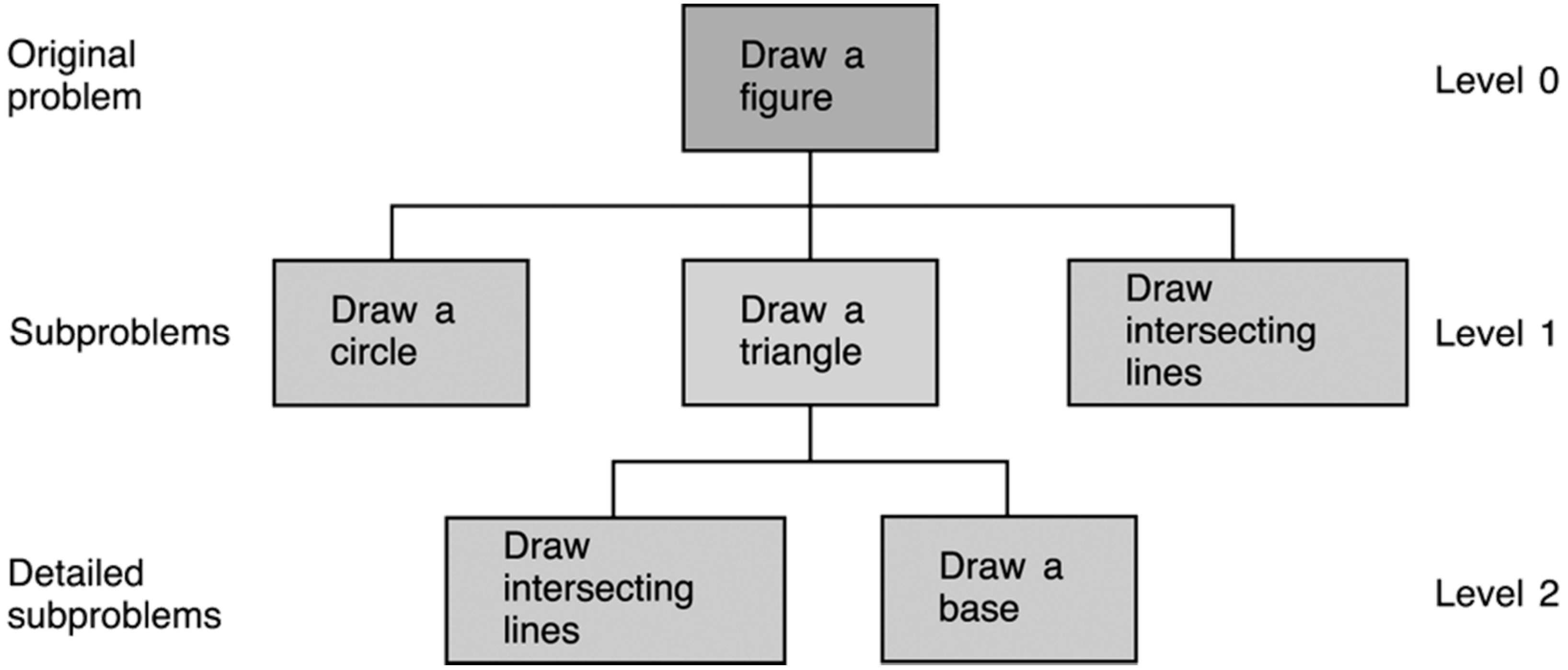
  Simple

  Diagrams



STRUCTURE CHARTS

- Structure Charts show the relationship between the original problem and its sub-problems.
- The sub-problem (Draw a triangle) can also be refined. It has its own sub-problems at level 2.



Functions Without Arguments

- One way to achieve top-down design is to **define a function** for each sub-program.
- For example, one can define functions to draw a circle, intersecting lines, base line, and a triangle.
- To draw a circle, call the function:

```
draw_circle();    /* No argument */
```

- To draw a triangle, call the function:

```
draw_triangle(); /* No argument */
```

- The above draw functions have **no arguments**

Function Prototypes

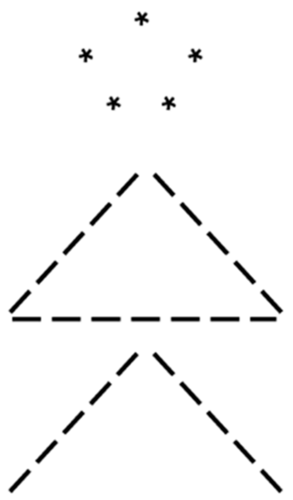
- A function must be declared before it can be used in a program.
- To do this, you can add a **function prototype** before **`main`** to tell the compiler what functions you are planning to use.
- A function prototype tells the C compiler:
  1. The result data type that the function will return
  2. The function name
  3. Information about the arguments that the function expects
- **Function prototypes** for **`draw_circle`** and **`sqrt`**

```
void   draw_circle(void);
double sqrt(double x);
```

Function Prototypes and Main Function

```
/* Draws a stick figure */

#include <stdio.h>

/* Function prototypes */
void draw_circle(void);           /* Draws a circle                          */

void draw_intersect(void);        /* Draws intersecting lines                */

void draw_base(void);             /* Draws a base line                       */

void draw_triangle(void);         /* Draws a triangle                        */

int
main(void)
{

      /* Draw a circle.                     */
      draw_circle();

      /* Draw a triangle.                   */
      draw_triangle();

      /* Draw intersecting lines.           */
      draw_intersect();

      return (0);
}

```

Draws This Stick Figure



FUNCTION DEFINITION

- A **function prototype** tells the compiler what arguments the function takes and what it returns, **but NOT what it does**
- A **function definition** tells the compiler **what the function does**
  - **Function Header:** Same as the prototype, except it does not end with a semicolon **;**
  - **Function Body:** enclosed by **{** and **}** containing variable declarations and executable statements

No Result

No Argument

```
/*
 * Draws a circle
 */
void
draw_circle(void)
{
      printf("   *   \n");
      printf(" *   *\n");
      printf("  * * \n");
}
```

```
/*
 * Draws a triangle
 */
void
draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

Placement of Function Definitions After the Main Function of a Program

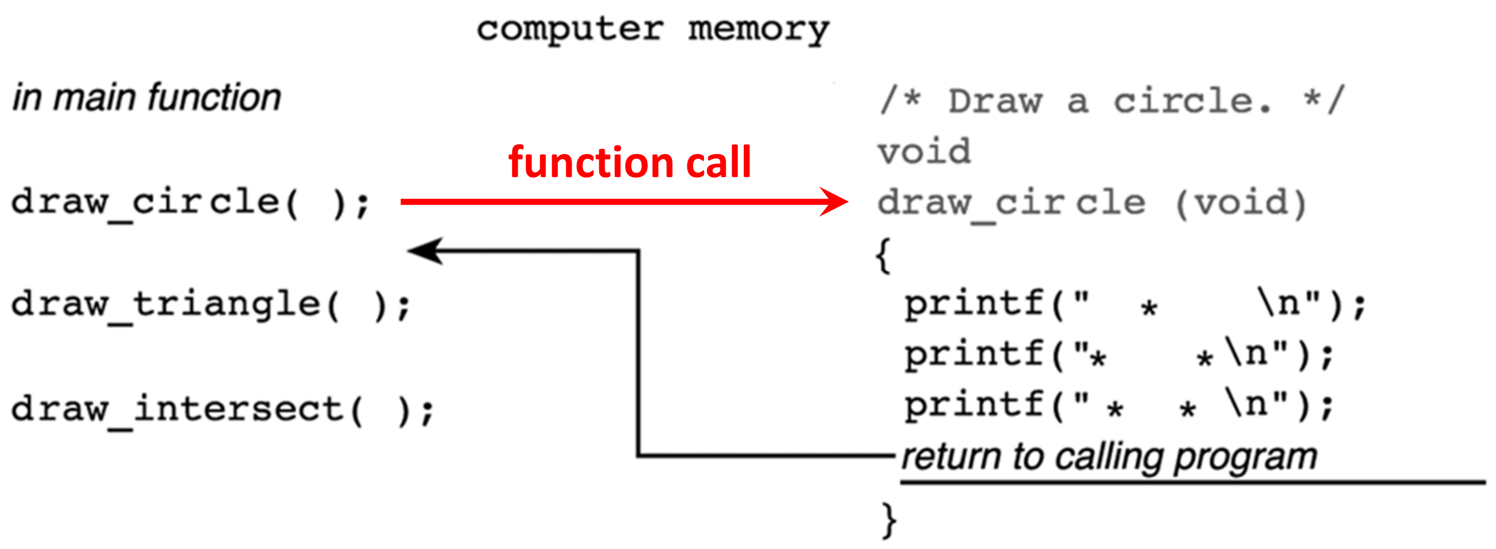
```c
/*
 * Draws a circle
 */
void
draw_circle(void)
{
      printf("   *   \n");
      printf(" *   * \n");
      printf("  * *  \n");
}

/*
 * Draws intersecting lines
 */
void
draw_intersect(void)
{
      printf("  / \\  \n"); /* Use 2 \'s to print 1 */
      printf(" /   \\ \n");
      printf("/     \\\n");
}

/*
 * Draws a base line
 */
void
draw_base(void)
{
      printf("-------\n");
}

/*
 * Draws a triangle
 */
void
draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

PLACEMENT OF FUNCTIONS IN A PROGRAM

- In general, declare all function prototypes at the beginning (after `#include` and `#define`)
- This is followed by the `main` function
- After that, we define all of our functions
- However, this is just a convention
- As long as a function's prototype appears before it is used, it doesn't matter where in the file it is defined
- The order we define functions in a program does not have any impact on how they are executed

EXECUTION ORDER OF FUNCTIONS

- Program execution always starts in `main` function
- Execution order of functions is determined by the **order of the function call** statements
- At the end of a function, control returns immediately after the point where the function call was made



NEXT . . .

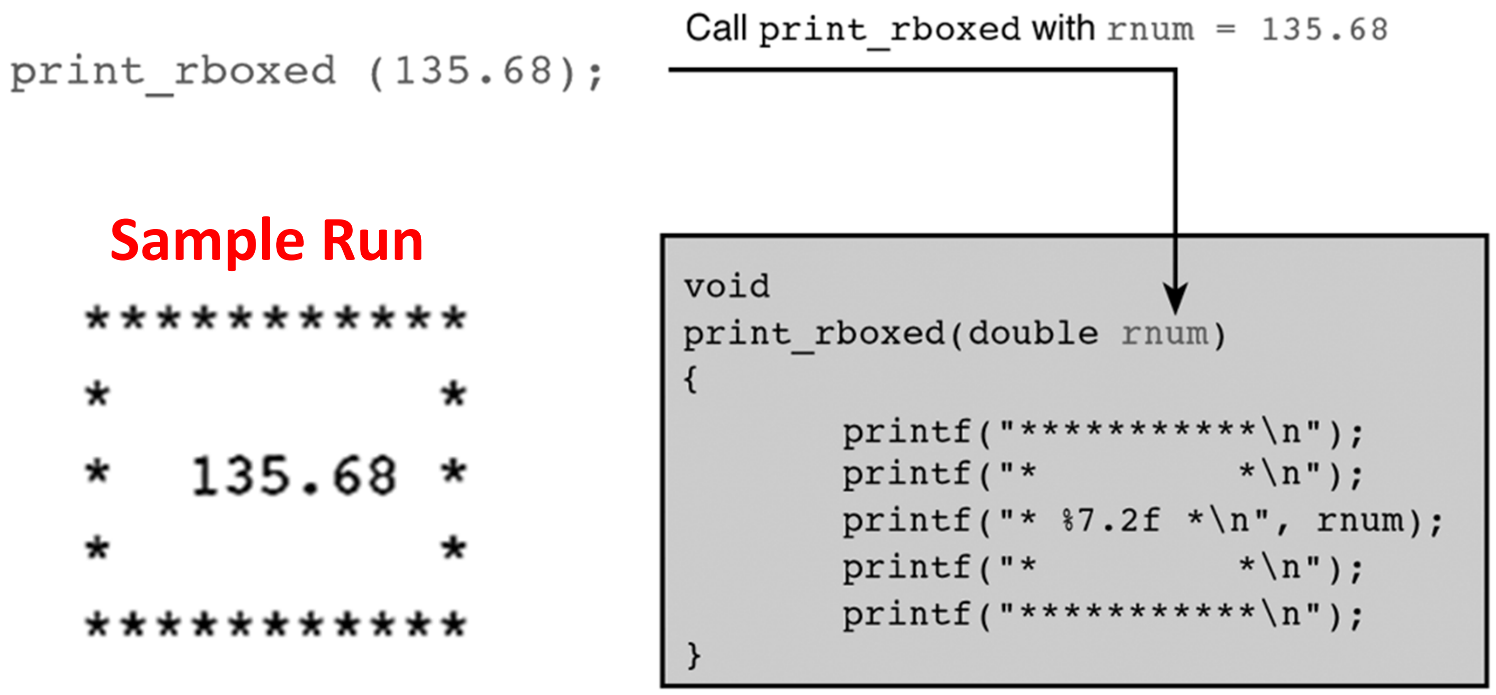
- Building Programs from Existing Information
- Library Functions and Code Reuse
- Top-Down Design and Structure Charts
- Functions, Prototypes, and Definitions
- **Functions with Arguments**
- **Testing Functions and Function Data Area**
- **Advantages of Functions and Common Errors**

Functions with Arguments

- We use **arguments** to communicate with the function
- Two types of function arguments:
  - **Input arguments:** pass data from the caller **to the function**
  - **Output arguments:** pass results **from the function** back to the caller [chapter 6]
- Types of Functions
  - No input arguments and no value returned
  - Input arguments, but no value returned
  - Input arguments and single value returned
  - Input arguments and multiple values returned [chapter 6]

Function with Input Argument but No Return Value

- `void print_rboxed(double rnum);`
- Display its **`double`** argument **`rnum`** in a box
- **`void`** function ➔ No return value



```
print_rboxed (135.68);
```

```
void
print_rboxed(double rnum)
{
        printf("***********\n");
        printf("*         *\n");
        printf("* %7.2f *\n", rnum);
        printf("*         *\n");
        printf("***********\n");
}
```

Sample Run

```
***********
*         *
*  135.68 *
*         *
***********
```

FORMAL AND ACTUAL PARAMETERS

- **Formal Parameter**

  An identifier that represents a parameter in a function prototype or definition.

  Example: `void print_rbox(double rnum);`

  The formal parameter is `rnum` of type `double`

- **Actual Parameter (or Argument)**

  An expression used inside the parentheses of a function call

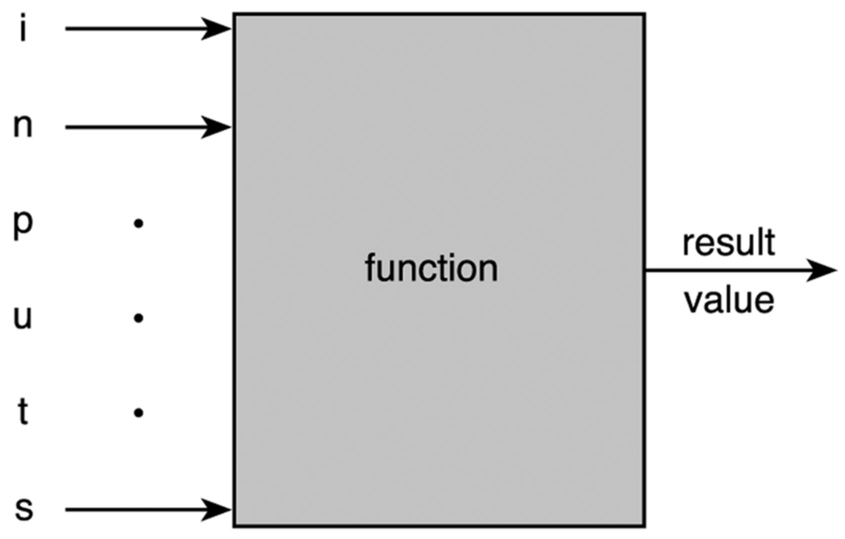
  Example: `print_rbox(x+y); /* function call */`

  Actual argument is the value of the expression `x+y`

- Parameters make functions more useful. Different arguments are passed each time a function is called.

FUNCTIONS WITH INPUT ARGUMENTS AND A SINGLE RESULT VALUE

```
/* area of a circle */
double
circle_area(double r)
{
  return (PI * r * r);
}

/* diagonal of rectangle */
double
rect_diagonal(double l, double w)
{
  double d = sqrt(l*l + w*w);
  return d;
}
```



- Functions in the math library are of this category

Testing Functions Using Drivers

- A function is an independent program module
- It should be tested separately to ensure correctness
- A **driver function** is written to test another function
  - Input or define the arguments
  - Call the function
  - Display the function result and verify its correctness
- We can use the `main` function as a driver function

TESTING FUNCTION `rect_diagonal`

```
/* Testing rect_diagonal function */
int
main(void)
{
  double length, width;   /* of a rectangle */
  double diagonal;        /* of a rectangle */

  printf("Enter length and width of rectangle> ");
  scanf("%lf%lf", &length, &width);
  diagonal = rect_diagonal(length, width);
  printf("Result of rect_diagonal is %f\n", diagonal);
  return 0;
}
```

ARGUMENT LIST CORRESPONDENCE

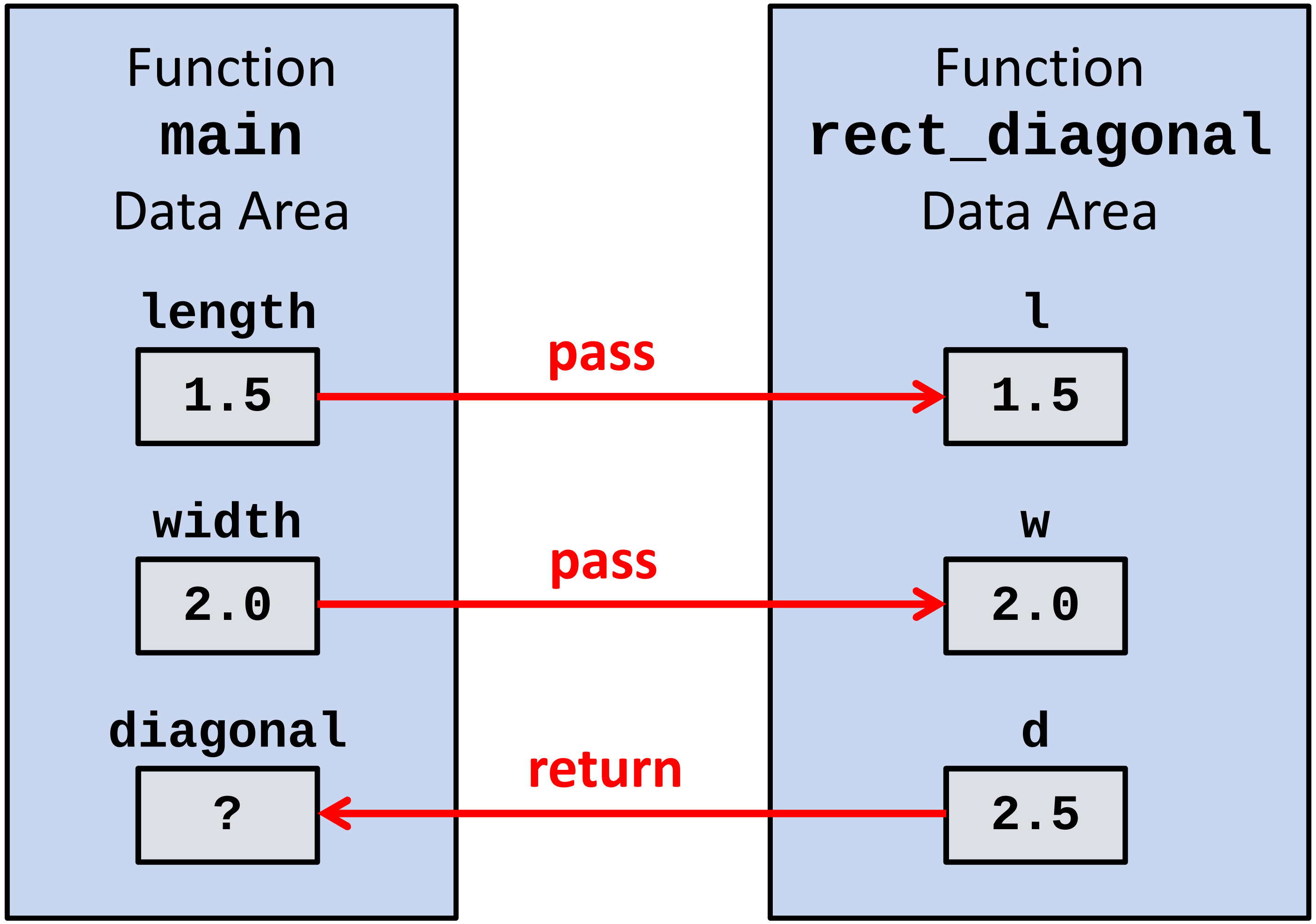
- The **Number** of actual arguments used in a call to a function must be equal to the number of formal parameters listed in the function prototype.
- The **Order** of the actual arguments used in the function call must correspond to the order of the parameters listed in the function prototype.
- Each actual argument must be of a data **Type** that can be assigned to the corresponding formal parameter with no unexpected loss of information.

The Function Data Area

- Each time a function call is executed, an area of memory is allocated for formal parameters and local variables
- **Local Variables**: variables declared within a function body
- **Function Data Area: Formal Parameters + Local Variables**
  - Allocated when the function is called
  - Can be used only from within the function
  - No other function can see them
- The function data area is **lost** when a **function returns**
- It is **reallocated** when the function is **called again**

Example of Function Data Areas

```
diagonal = rect_diagonal(length, width);
```



Advantages of Functions

- A large problem can be better solved by breaking it up into several functions (sub-problems)
- Easier to write and maintain small functions than writing one large main function
- Once you have written and tested a function, it can be reused as a building block for a large program
- Well written and tested functions reduce the overall length of the program and the chance of error
- Useful functions can be bundled into libraries

Programming Style

- Each function should begin with a comment that describes its purpose, input arguments, and result
- Include comments within the function body to describe local variables and the algorithm steps
- Place prototypes for your own functions in the source file before the `main` function
- Place the function definitions after the `main` function in any order than you want

COMMON PROGRAMMING ERRORS

- Remember to use **`#include`** directive for every standard library from which you are using functions
- For each function call:
  - Provide the required **Number** of arguments
  - Make sure the **Order** of arguments is correct
  - Make sure each argument is the correct **Type** or that conversion to the correct type will not lose information.
- Document and test every function you write
- Do not call a function and pass arguments that are out of range. A function will not work properly when passing invalid arguments: **`sqrt(-1.0)`**