ICS-103

Computer Programming in C

Lectures 7-9

Chapter **3:**

Top-Down Design with Functions

Dr. Tarek Ahmed Helmy El-Basuny

Outline of Ch. 3 Topics

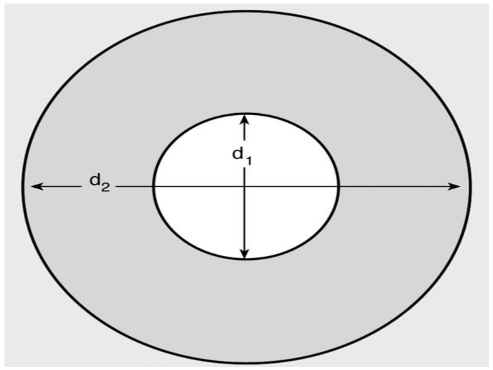
- Building Programs from Existing Information
- Library Functions and Code Reuse
- Top-Down Design and Structure Charts
- Introduction of Functions in C
- Function Prototype, and Definition
- Functions Without Arguments
- Placement of Functions in a Program
- Execution Order of Functions in the C program
- Functions with Arguments
  - Function with Input Argument but no Return Value
  - Functions with Input Arguments and a Single Return Value
- Formal and Actual Parameters
- Testing Functions
- Function Data Area
- Advantages of Functions
- Common Programming Errors with Functions call
- Problem Solving Examples

Recall: Software Development Steps

1. Specify/Identify the problem,
2. Analyze/Understand the problem (input and output),
3. Design the algorithm to solve the problem,
4. Implement/Code the algorithm using a programming language,
5. Test and verify the implemented program,
6. Maintain and update the program.

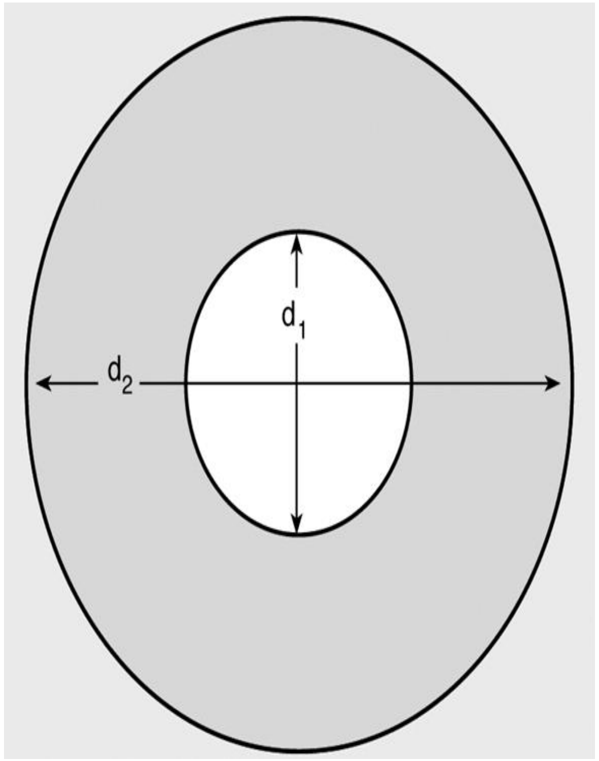
Case Study: Computing the Weight of a Batch of Flat Washers

- Problem: assume that you work for a hardware company that manufactures flat washers. To estimate the shipping costs, your company asked you to write a C program that computes the weight of a specified quantity of flat washers.
- Analysis: the flat washer resembles a small donut. To compute the **weight** of a single flat washer, you need to know is rim-area, its thickness, and the density of the material used in its construction.



- Inputs:
  - Hole diameter, edge diameter, thickness,
  - Density of the material, quantity of the washers made.
- Output:
  - Weight (of a batch of flat washers)

2. Analysis: Computing the Area and Weight



rim area = area of the outer circle- area of the hole (inner circle)

rim area = $\pi(d_2/2)^2 - \pi(d_1/2)^2$

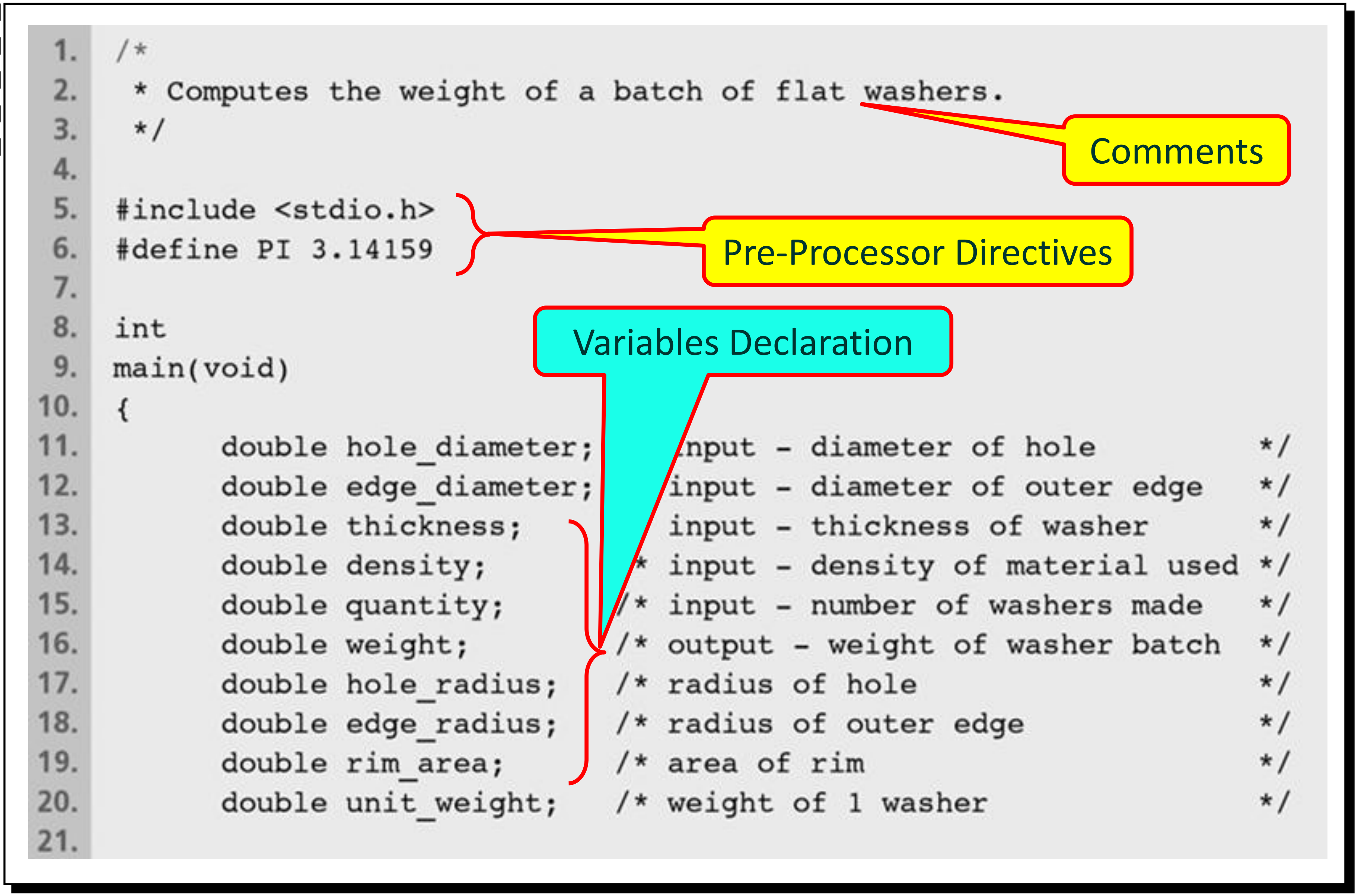
unit weight = *rim area* × *thickness* × *density*

Total weight = *unit weight* × *quantity*

3. Designing The Algorithm

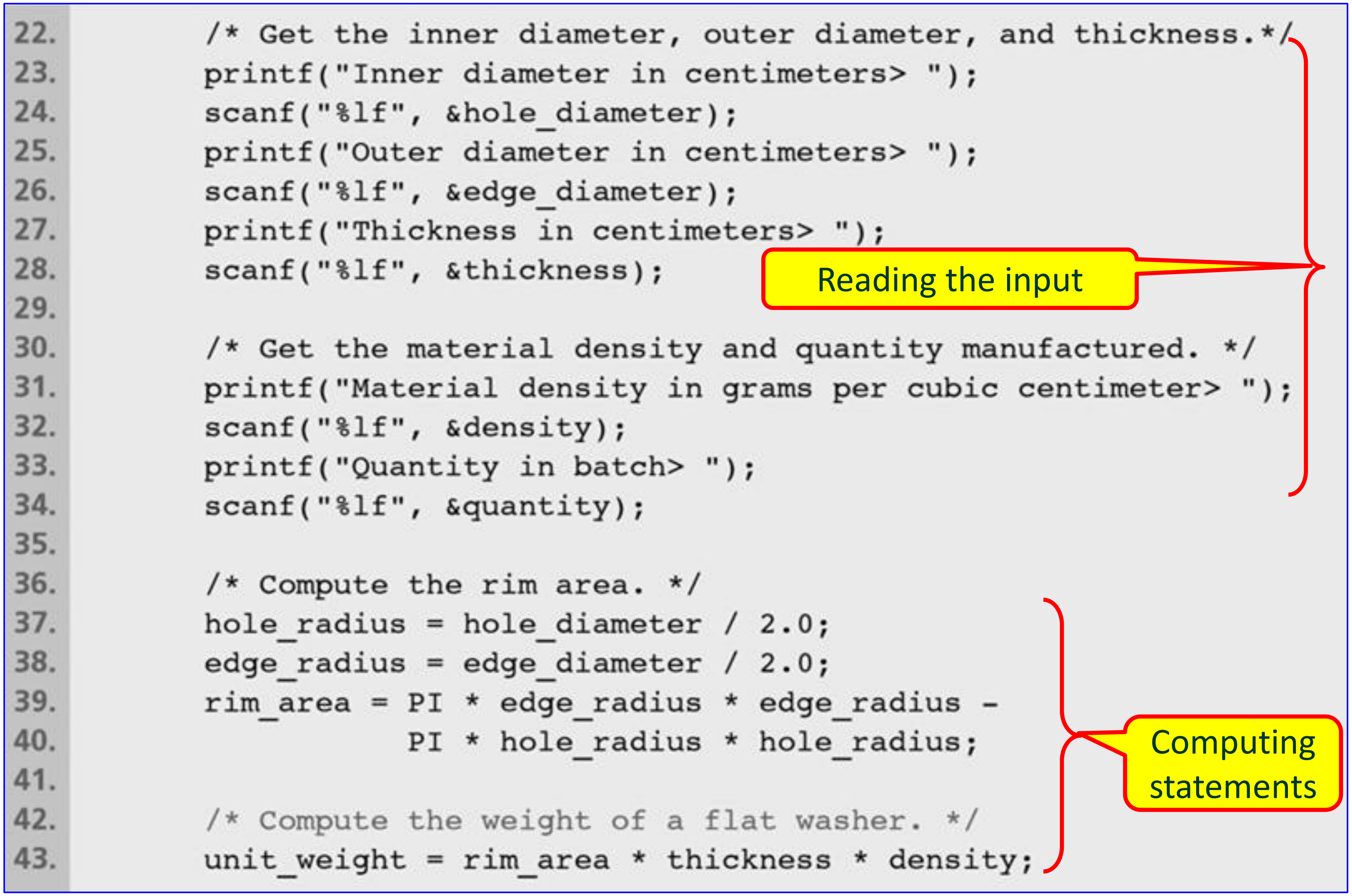
1. Read the washer's inner diameter, outer diameter, and thickness.
2. Read the material density and quantify of washers.
3. Compute the rim area.
4. Compute the weight of one flat washer.
5. Compute the weight of the batch of washers.
6. Display the total weight of the batch of washers.

4. Implement Flat Washer C Program



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

#include <stdio.h>
#define PI 3.14159

int
main(void)
{
      double hole_diameter;      nput - 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              */

```

Flat Washer Program (Cont'd)

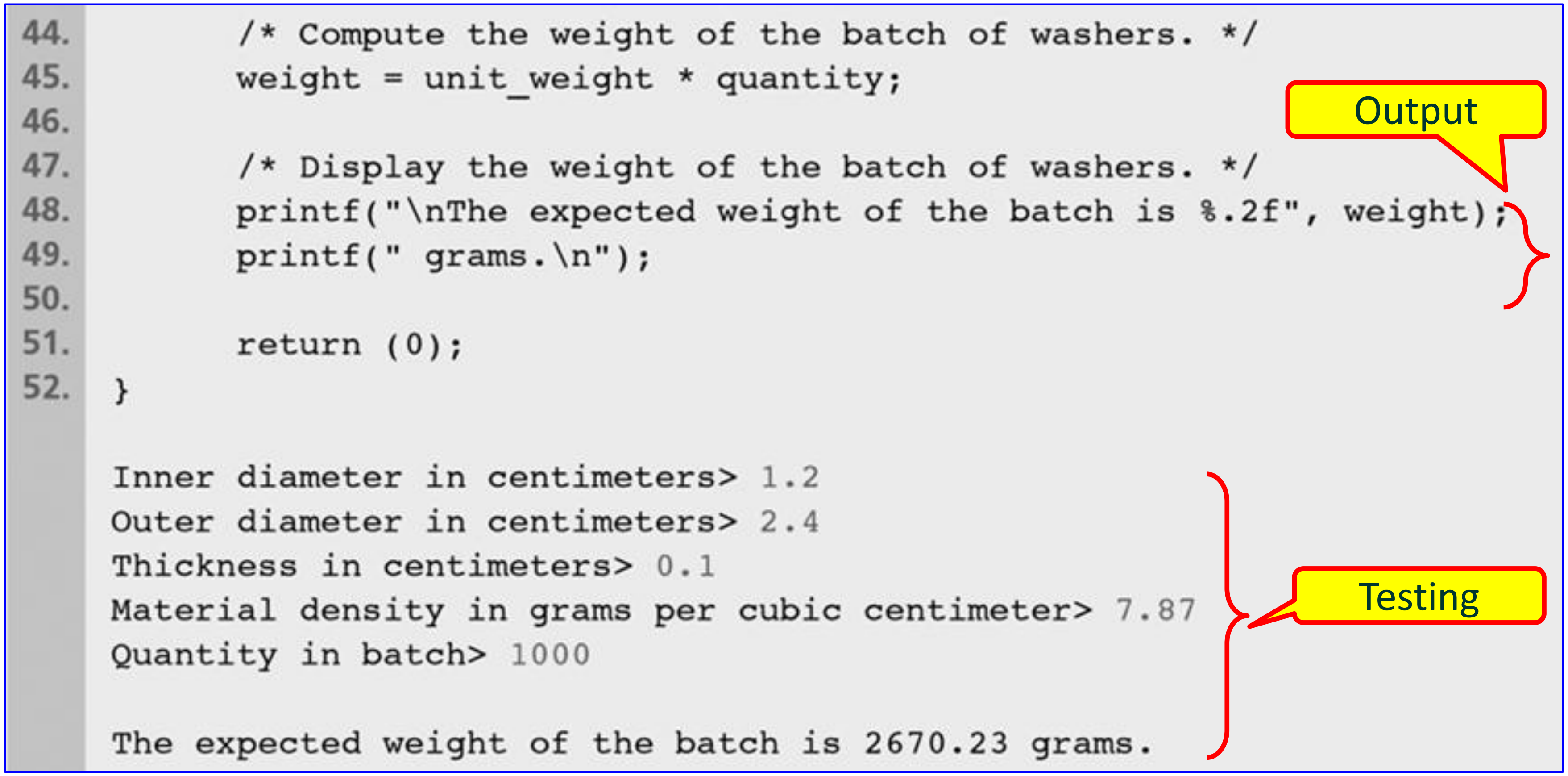


```
/* Get the inner diameter, outer diameter, and thickness.*/
printf("Inner diameter in centimeters> ");
scanf("%lf", &hole_diameter);
printf("Outer diameter in centimeters> ");
scanf("%lf", &edge_diameter);
printf("Thickness in centimeters> ");
scanf("%lf", &thickness);

/* 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;
```

Reading the input

Computing statements

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 le ad to calculations of the weight.
- Verify the correctness of the weight.

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** by professional programmers is one way to achieve this.
- C promotes code reuse by providing library functions. i.e.
  - Input/Output functions: printf, scanf , etc.
  - Mathematical functions: sqrt, exp, log, etc.
  - String functions: strlen, strcpy, strcmp, etc.
- Each of the standard library function can be called by writing its name and passing to it the required arguments.
  - i.e. y=sqrt (x);
- Next slide show some of the C standard library functions.
- The appendix of the text book lists all C standard library functions.

Some Mathematical Library Functions

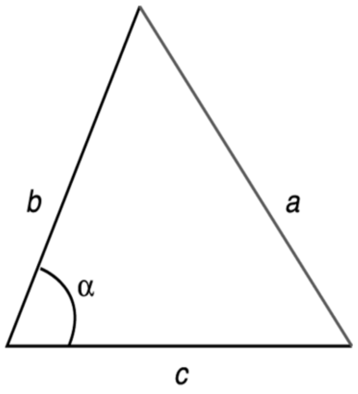
Function	Header file	Argument	Result	Example
abs(x)	<stdlib.h>	int	int	abs(-5) is 5
fabs(x)	<math.h>	double	double	fabs(-2.3) is 2.3
sqrt(x)	<math.h>	double	double	sqrt(2.25) is 1.5
exp(x)	<math.h>	double	double	exp(1.0) is 2.71828
log(x)	<math.h>	double	double	log(2.71828) is 1.0
log10(x)	<math.h>	double	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

- We can use the C math function (#include <math.h>) to calculate roots of a quadratic equation $ax^2 + bx + c = 0$
  - `root1 = (-b + sqrt(b*b – 4*a*c))/(2.0 * a);`
  - `root2 = (-b - sqrt(b*b – 4*a*c))/(2.0 * a);`
  - We can use `pow(b,2)` to calculate `b*b;\`
  - `delta = pow(b,2)- 4*a*c;`
  - `root1 = (-b + sqrt(delta))/(2.0 * a);`
  - `root2 = (-b - sqrt(delta))/(2.0 * a);`
- We can use C math function(#include<math.h>) to compute the unknown side of a triangle
- $a^2 = b^2 + c^2 – 2\, b\, c \cos(\alpha)$

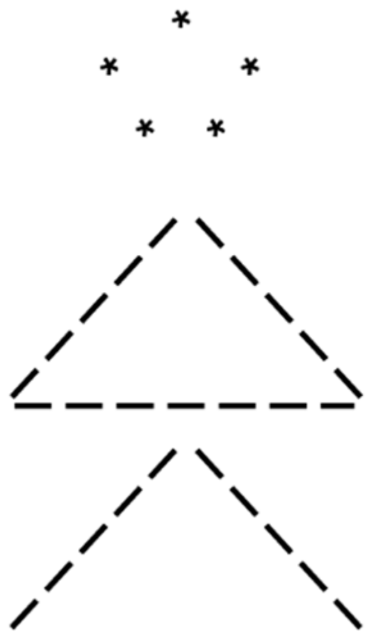
```
a = sqrt(b*b + c*c -
     2*b*c*cos(alpha));
```

- alpha must be in radians



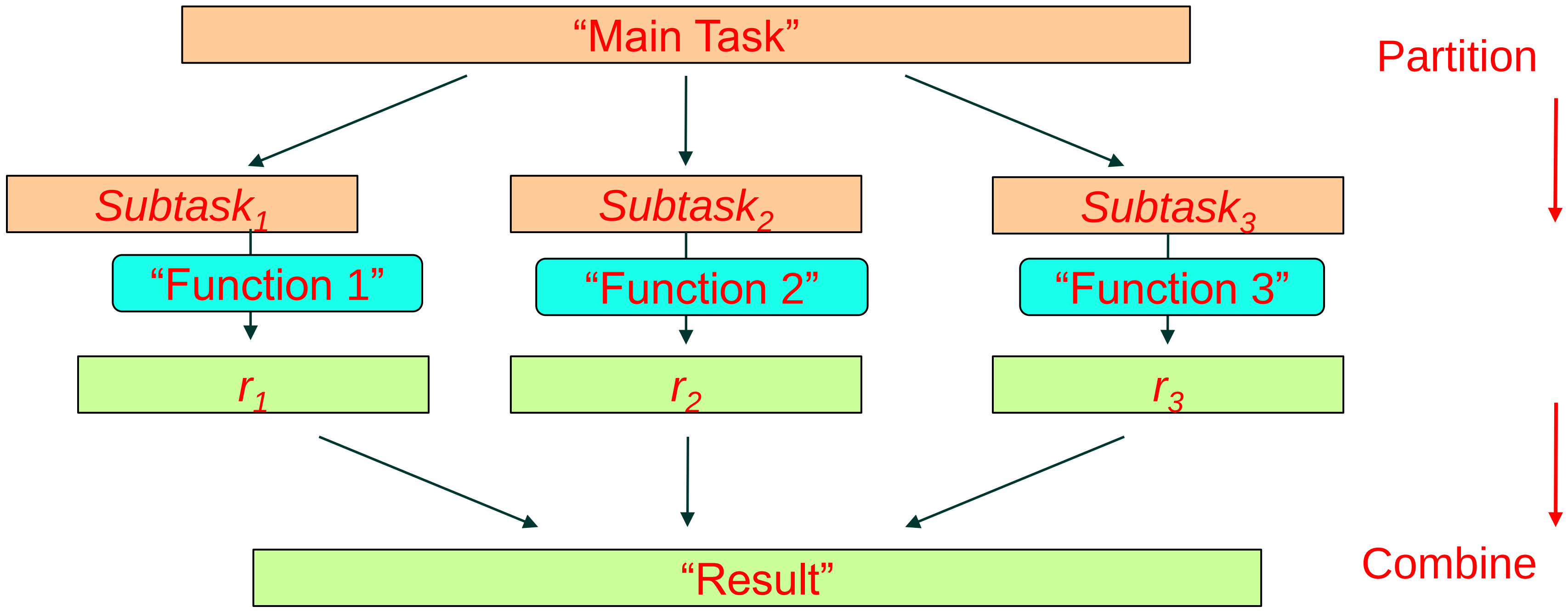
Top-Down Design

- Algorithms needed to solve problems are often complex.
- To solve a problem, **<u>the programmer must break it into simple sub-problems at a lower level</u>**.
- This process is called top-down design.
- Example:
  - You have been asked to draw a simple diagram that consists of a circular shape, a triangle shape and a triangle without its base. you can divide the problem into three sub-problems.



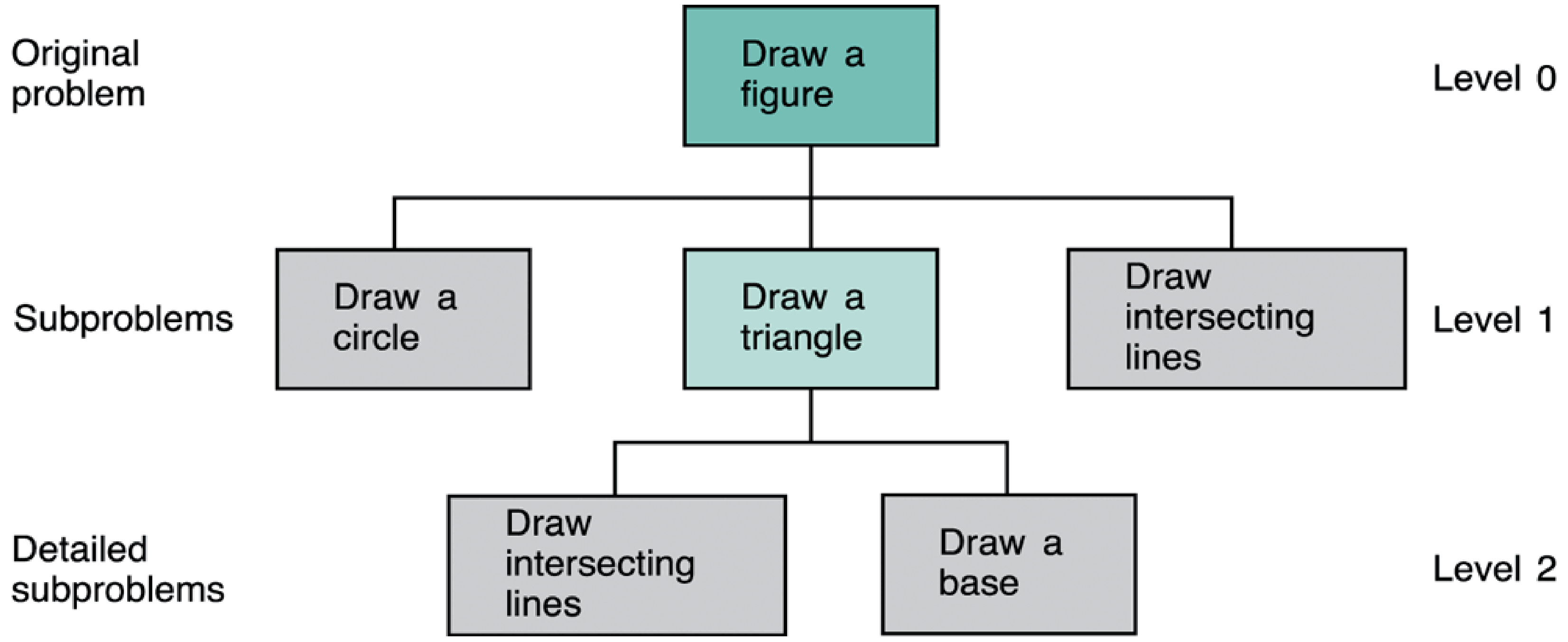
Divide and Conquer

- If the problem is complex, then divide it into sub-problems that we can handle:
  - Applies the concept of abstraction
  - The divide-and-conquer approach can be applied over and over again until each subtask is manageable.



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 to its own sub-problems at level 2.



Top-Down Design

- **One way to achieve top-down design** is to **define a function for each sub-program** and then call them from the main function.
- 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.**

Introduction to Functions in C

- A **function** is a group of statements that together perform a task.
- A **function** accepts zero or more argument values, produces a result value (it may do nothing), and returns zero or more results.
  - Functions can be written and tested separately
  - Functions can be reused by other programs
- All functions defined at “top level” of C programs
  - Functions in C do not allow other functions to be declared within them
    - **We can not nest functions in C**.
  - Can be linked by any other program that knows the function prototype
- Functions could be
  - Pre-defined library functions (e.g., printf, sin, sqrt, etc.) or
  - Programmer-defined functions (e.g., factorial, draw_circle, etc.)

Example: Pre-defined library Functions

#include <math.h>

- **sin(x)** // radians
- **cos(x)** // radians
- **tan(x)** // radians
- **atan(x)**
- **atan2(y,x)**
- **exp(x)** // e^x
- **log(x)** // $\log_e x$
- **log10(x)** // $\log_{10} x$
- **sqrt(x)** // $x \geq 0$
- **pow(x, y)** // x^y
- **...**

#include <stdio.h>

- **printf()**
- **fprintf()**
- **scanf()**
- **sscanf()**
- **...**

#include <string.h>

- **strcpy()**
- **strcat()**
- **strcmp()**
- **strlen()**
- **...**

Function Definitions

- **Function definition format**

```
        return-value-type  function-name( parameter-list )
          {
            declarations and statements
          }
```

- **Function-name**: any valid identifier
- **Return-value-type**: data type of the result (the default type is **int**)
  - **void** indicates that the function returns nothing.
- **Parameter-list**: comma separated list of declares parameters.
  - A type must be listed explicitly for each parameter unless, the parameter is of type **int.**
  - **void can be used if there is no parameters.**
- Declarations and statements: {Function body}
  - Local variables can be declared inside the braces {Function body}
  - Functions **can not** be defined inside other functions.
  - Returning control
    - If nothing returned
      - **return;** or, until reaches right brace
    - If something returned
      - **return** *expression*;

Nesting Functions

- Can we Nest functions (can we place the code for one function inside another function)?
  - ➲ The answer is NO!
- We **can** utilize (call) other functions inside a function, but we cannot create a new function there.
- ***// Here is pseudo code of the correct layout of two functions***

```
function1()
{
  code;
  code;
  code;
}

function2()
{
  code;
  code;
  code;
}
```

// **INCORRECT** layout of two functions

```
function1()
{
  code;
  code;
  code;
  function2()
{
  code;
  code;
  code;
}
}
```

Example: User-defined Function

```
return_type function_name (parameters)
{
      declarations;
      statements;
}
```

```
int my_add_func(int a, int b)

{

   int sum;

   sum = a + b;

   return sum;

}
```

Function Definition and Call: Example

- **A function definition has the following syntax**:

```
type function name(parameter list) {
        local declarations
        sequence of statements
}
```

- **For example**: Definition of a function that computes the absolute value of an integer:

```
int absolute(int x)
    {
         if (x >= 0)   return x;
         else
           return -x;
     }
```

- **A function call has the following syntax:**

  function name(argument list);

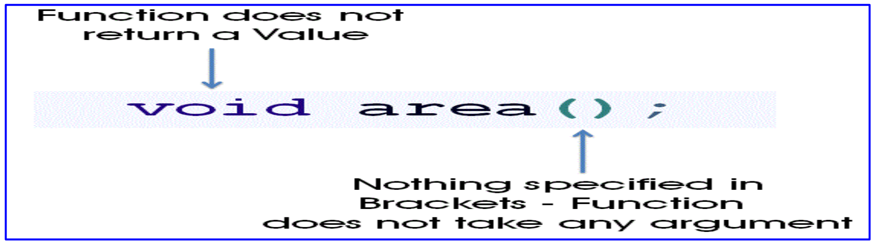
  Example: int distance = absolute(-5);

  ➲ The result of a function call is a value of type <**type**>

Functions Without Arguments

- Functions with no parameters (arguments) can be used in C. Usually they will not return a value but carry out some operation.
- Example:

```
#include<stdio.h>
void area();            // Prototype Declaration
void main()
{
area();   //function call
}
void area() //function definition
{
      float area_circle;
      float rad;
      printf("\nEnter the radius: ");
      scanf("%f", &rad);
      area_circle = 3.14 * rad * rad ;
      printf("Area of Circle = %f",area_circle);
}
```


Function does not return a Value
void area ( ) ;
Nothing specified in Brackets - Function does not take any argument

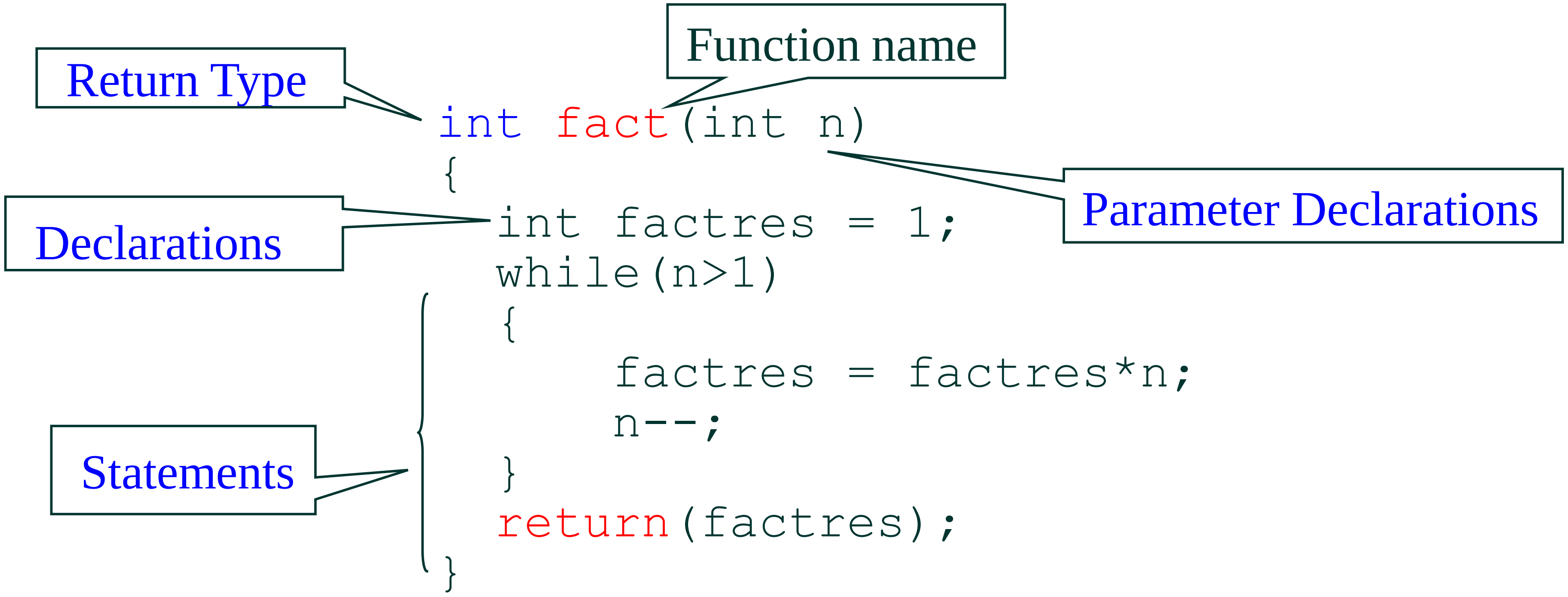
Output

```
Enter the radius : 3
Area of Circle = 28.260000
```

Value Returning Functions

- Function *returns* a single value to the calling program.
- Function definition declares the type of value to be returned.
- A return *expression;* statement is *required* in the function definition
- The value returned by a function **can be assigned to a variable, printed, or used in an expression**.

n!=n*(n-1)*…*1, 0! = 1 by definition



```
int fact(int n)
{
   int factres = 1;
   while(n>1)
   {
         factres = factres*n;
         n--;
   }
   return(factres);
}
```

Void Functions

- A **void function** does not return a value to the calling program
- A **void function** may be called to
  - Perform a particular task (clear the screen)
  - Modify data
  - Perform input and output
- A **return**; statement can be used to exit from function without returning any value.

- Write a program to generate the following output?

```
*
**
***
****
*****
```

```
for (i=1; i<=5; i++) {
  for (j=1; j<=i; j++)
      printf("*");
  printf("\n");
}
```

```
#include <stdio.h>
void print_i_star(int i);
main()
{
  int i;
  for (i=1; i<=5; i++) {
     print_i_star( i );
  }
}
void print_i_star(int i)
{
  int j;
  for (j=1; j<=i; j++)
      printf("*");
  printf("\n");
  return;
}
```

Function Prototype

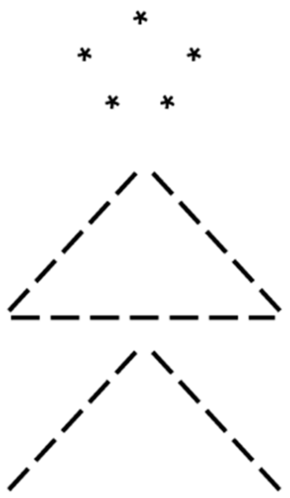
- A function must be declared before it can be used in a program.
- A function prototype tells the C compiler:
  1. The result data type that the function will return.
  2. The function name.
  3. Information about the arguments that the function expects.
- **Example:** double myfunction(int n);
  - This prototype specifies that in this program, there is a function named "myfunction" which takes a single integer argument "n" and returns a double.
- A function prototype tells the compiler what arguments the function takes and what it returns, but NOT what it does.
- Usually function prototypes should be before the main to tell the compiler what functions you are planning to use.
- Function prototypes for draw_circle and sqrt:
  - void draw_circle(void);
  - double sqrt(double x);

Function Prototypes

```c
/*
 * Draws a stick figur
 */

#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);
}
```

Function Prototypes
Before main function

Draws
This
Stick
Figure



Function Definition

- 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");
}
```

Placement of Function Definitions After the Main Function Of a Program

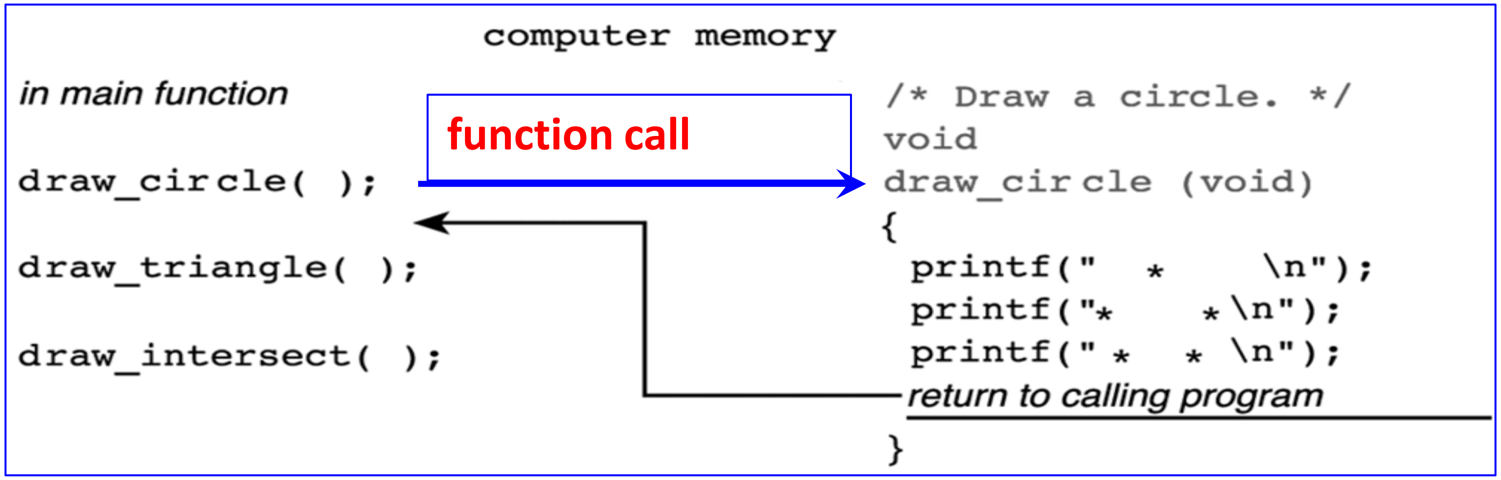
```
/*
 * Draws a base line
 */
void
draw_base(void)
{
        printf("--------\n");
}

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

Placement of Functions in a Program

- A **function in C** is a block of code that takes as input (parameters values) from the caller function, does some computation, and (usually) returns the result to the caller function, or it may return nothing.
- **Declare all function prototypes at the top** (after #include and #define).
  - Function prototype In C, specifics RETURN_TYPE, **Name_of_Function** and (the list of parameters);
  - This information is communicated to the compiler for later usage.
- This is followed by the main function.
  - The main function is always the first code executed when a program starts.
- Define all of the required functions after the main function.
  - As long as a function's prototype appears before it is used, it doesn't matter where in the file it is defined.
- **Ordering of functions in a program**
  - The order of functions inside a program is arbitrary. It **does not matter** if you put function one at the top of the file and function two at the bottom, or vice versa.
- 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.
- When a function is "called" the program "leaves" the current section of code and begins to execute the first line inside the function.
- **Thus the functions "flow of control" is**:
  - The program comes to a line of code containing a "function call".
  - The program enters the function (starts at the first line in the function code).
  - All instructions inside of the function are executed from top to bottom.
  - After executing of a function, the control goes back to where it started from.
  - Any data computed and RETURNED by the function is used in place of the function in the original line of code.



```
draw_circle( );

draw_triangle( );

draw_intersect( );
```

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

Execution Order of Functions

- Execution order of functions is determined by the order of the function call statements.

```
main() {
display();
}
void mumbai() {
printf("In mumbai");
}
void pune() {
india();
}
void display() {
pune();
}
void india() {
mumbai();
}
```

- In this program we have written functions in the following order:-
  - main()
  - mumbai()
  - pune()
  - display()
  - india()

- However functions are called in the order we call them as following:-
  - main()
  - display()
  - pune()
  - india()
  - mumbai().

Functions with Arguments

- We use arguments to communicate with the function.

- **Two types of function arguments**:
  - Input arguments: pass data from the caller (main) to the called function.
  - Output arguments: pass results from the called function back to the caller [will be discussed in chapter 6].

- **Types of Functions:**
  1. No input arguments and no value returned, we discussed before.
  2. Input arguments, but no value returned, will be discussed today.
  3. Input arguments and single value returned, will be discussed today.
  4. Input arguments and multiple values returned [will be discussed in chapter 6].

Function with Input Argument but no Return Value

- The function which accepts argument but it does not return a value back to the calling function.
  - It is (One-way) type communication function.
  - Generally Output is printed in the called function.
  - Here area is called function and main is calling function.
- **Example:**

```
#include<stdio.h>
#include<conio.h>
void area(float rad); // Prototype Declaration
main()  {    // main function
float rad;    // variable declaration
printf("nEnter the radius : ");
scanf("%f",&rad);
area(rad);  // function call
getch();
}
void area(float rad) { //area function with input argument but no return value
float ar;
ar = 3.14 * rad * rad ;
printf("Area of Circle = %f",ar);
}
```

Formal and Actual Parameters

- When we create a function, it should represent a "generic" form that can be applied in many circumstances.
- Formal Parameter
  - **An identifier that represents a parameter in a function prototype or definition**: i.e:
    - void print_rbox(double rnum);
    - double average_grade (list_of_grades) { body }
    - The **formal parameters** are rnum and list_of_grades
- Actual Parameter (or Argument)
  - A value/expression used inside the parentheses of a function call: i.e.:

```
print_rbox(5);  // function call
midterm_grades = ... // if we created array of grades
print average_grade(midterm_grades)
```

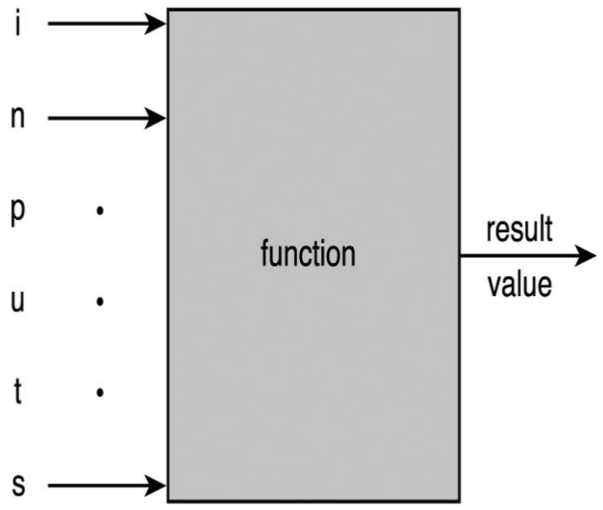
- Actual arguments are the value of the formal parameters (5 and midterm_grades)
- Formal parameters must match with actual parameters in *order*, *number* and *data type*.
- If the type is not the same, type conversion will be applied. But this might cause some errors (double→int) so you need to be careful!

Functions with Input Arguments and a Single Result Value

- Functions accepts argument and returns a value back to the calling function.
- It can be termed as **Two-way Communication** between calling function and called function.

Example:

```
/* 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.
- **Driver:**
  - A short function written to test another function by defining its arguments, calling it, and displaying its result.
- 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); /* read multiple values*/
  diagonal = rect_diagonal(length, width); /* call to test */
  printf("Result of rect_diagonal is %f\n", diagonal);
  return 0;
}
```

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

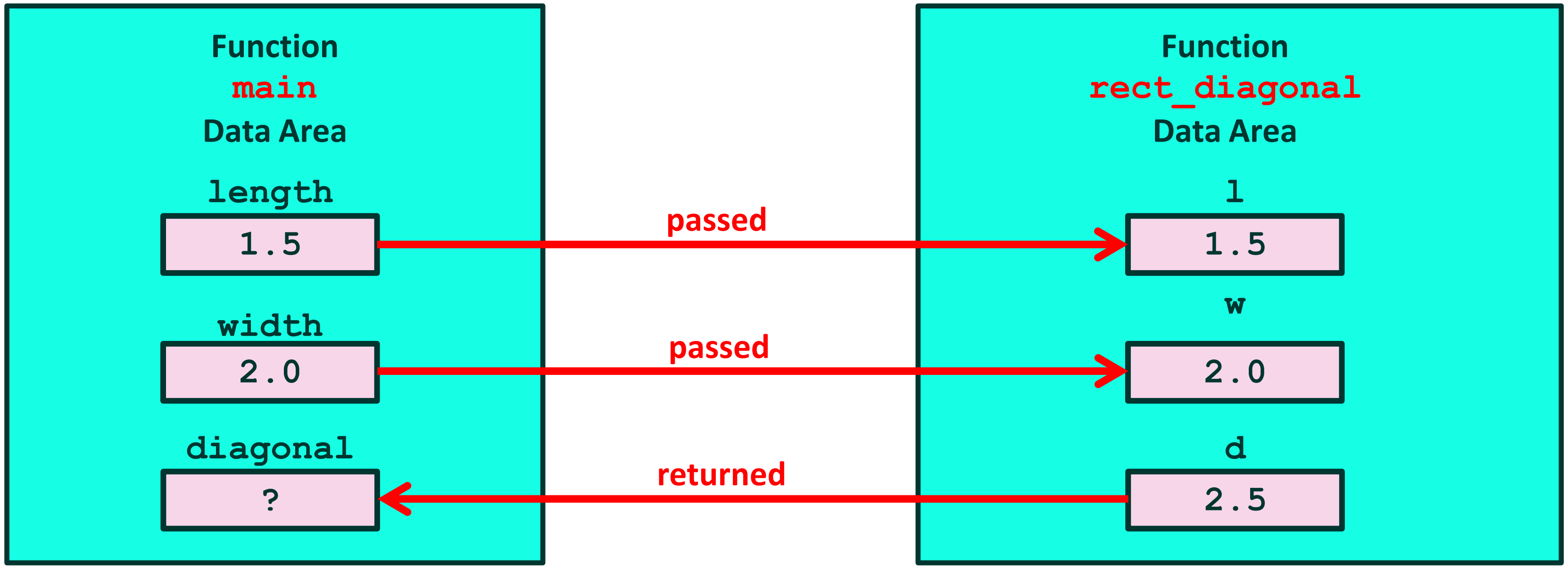
Outline of Ch. 3 Topics

- **In the Last class, we discussed:**
  - Building Programs from Existing Information
  - Library Functions and Code Reuse
  - Top-Down Design and Structure Charts
  - Introduction of Functions in C
  - Function Prototype, and Definition
  - Functions Without Arguments
  - Placement of Functions in a Program
  - Execution Order of Functions in the C program
  - Functions with Arguments
    - Function with Input Argument but no Return Value
    - Functions with Input Arguments and a Single Return Value
  - Formal and Actual Parameters
  - Testing Functions
- **In today's class, we will discuss:**
  - Function Data Area
  - Advantages of Functions
  - Common programming Errors with Functions call

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



```
Int main(void)
{ double length, width; diagonal;
  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;
}
```

```
double rect_diagonal(double l, double w)
{
  double d = sqrt(l*l + w*w); //d is local variable
  return d;
}
```

Argument List Correspondence (NOT rule)

- 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.

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/archived into libraries and reused.

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.**
- **<u>Place the function definitions after the `main` function in any order that you want</u>.**

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.
  - I.e. calling sqrt(-1.0) with -1.0 as argument will cause an error.

Problem Solving 1: Formulating Algorithm with Top-Down

- The problem is:
  - Develop a **class-averaging program** that will:
    - Process an arbitrary number of grades each time the program runs and calculates the average score.
  - The number of students is unknown.
  - How will the program know to end?

- We can use sentinel/flag value to **Indicates "end of data entry."**
  - Loop ends when user inputs the sentinel value.
  - Sentinel value chosen so it cannot be confused with a regular input (such as -1 in this case).

Problem Solving 1: Formulating Algorithm with Top-Down

- Top-down design
  - Begin with a pseudo-code representation of the top:

    Determine the class average for the quiz
  - Divide top into smaller tasks and list them in order:

    Initialize variables
    Input, sum and count the quiz grades
    Calculate and print the class average
- Programs have three phases:
  - Initialization: initializes the program variables.
  - Processing: inputs data values and adjusts program variables accordingly.
  - Termination: calculates and prints the final results.

Problem Solving 1: Formulating Algorithm with Top-Down

- Initialize variables as following:

```
Initialize total to zero
Initialize counter to zero
```

- Refine Input, sum and count the quiz grades to
    - Input the first grade (possibly the sentinel)
    - While the user has not as yet entered the sentinel
    - Add this grade into the running total
    - Add one to the grade counter
    - Input the next grade (possibly the sentinel)

- Refine Calculate and print the class average

```
If the counter is not equal to zero
   Set the average to the total divided by the counter
   Print the average.
else
   Print “No grades were entered”
```

Problem Solving 1: C Program

```c
/*
   Class average program with
   sentinel-controlled repetition */
#include <stdio.h>

int main()
{
   float average;                 /* new data type */
   int counter, grade, total;

   /* initialization phase */
   total = 0;
   counter = 0;

   /* processing phase */
   printf( "Enter grade, -1 to end: " );
   scanf( "%d", &grade );

   while ( grade != -1 ) {
      total = total + grade;
      counter = counter + 1;
      printf( "Enter grade, -1 to end: " );
      scanf( "%d", &grade );
   }
```

Problem Solving 1: C Program

```c

   /* termination phase */
   if ( counter != 0 ) {
      average = ( float ) total / counter;
      printf( "Class average is %.2f", average );
   }
   else
      printf( "No grades were entered\n" );

   return 0;   /* indicate program ended successfully */
}
```

- Calculate Average
- Print Results

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

- Program Output

Problem Solving 2: Formulating Algorithm with Top-Down

- **Problem**
  - A college has a list of test results (**1** = pass, **2** = fail) for 10 students.
  - Write a program that analyzes the results.
    - If more than 8 students pass, print "Raise Tuition"
- **Notice that**
  - The program must process 10 test results
    - That means counter-controlled loop will be used.
  - Two counters can be used
    - One for number of passes, and one for number of fails
  - Each test result is a number: either a `1` or a `2`
    - If the number is not a `1`, we assume that it is a `2`.

Problem Solving 2: Formulating Algorithm with Top-Down

- **Top level outline**
  - Analyze exam results and decide if tuition should be raised
- **Initialize variables**
  - Initialize passes to zero
  - Initialize failures to zero
  - Initialize student counter to one
- **Input the ten quiz grades and count passes and failures:**
  - While student counter is less than or equal to ten
    - Input the next exam result
  - If the student passed
    - Add one to passes
  - Else   Add one to failures
    - Add one to student counter
- **Print a summary of the exam results and decide if tuition should be raised:**
  - Print the number of passes
  - Print the number of failures
  - If more than eight students passed Print "Raise tuition"

Problem Solving 2: C Program

```c
   /* Analysis of examination results */
#include <stdio.h>

int main()
{
   /* initializing variables in declarations */
   int passes = 0, failures = 0, student = 1, result;

   /* process 10 students; counter-controlled loop */
   while ( student <= 10 ) {
      printf( "Enter result ( 1=pass,2=fail ): " );
      scanf( "%d", &result );

      if ( result == 1 )          /* if/else nested in while */
         passes = passes + 1;
      else
         failures = failures + 1;

      student = student + 1;
   }

   printf( "Passed %d\n", passes );
   printf( "Failed %d\n", failures );

   if ( passes > 8 )
      printf( "Raise tuition\n" );

   return 0;   /* successful termination */
}
```

- Initialize variables
- Input data and count passes/failures
- Print results

Problem Solving 2: C Program

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

- Program Output

The End!!

Thank you

Any Questions?