



# PROGRAMMING METHODOLOGY (PHƯƠNG PHÁP LẬP TRÌNH)

---

## UNIT 4: Top-Down Design & Functions

# Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

# Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

# Recording of modifications

- Currently, there are no modification on these contents.

# Unit 4: Top-Down Design & Functions

## Objectives:

- How to analyse, design, and implement a program
- How to break a problem into sub-problems with step-wise refinement
- How to create your own user-defined functions

## Reference:

- Chapter 5 Functions: Lessons 5.1 – 5.3

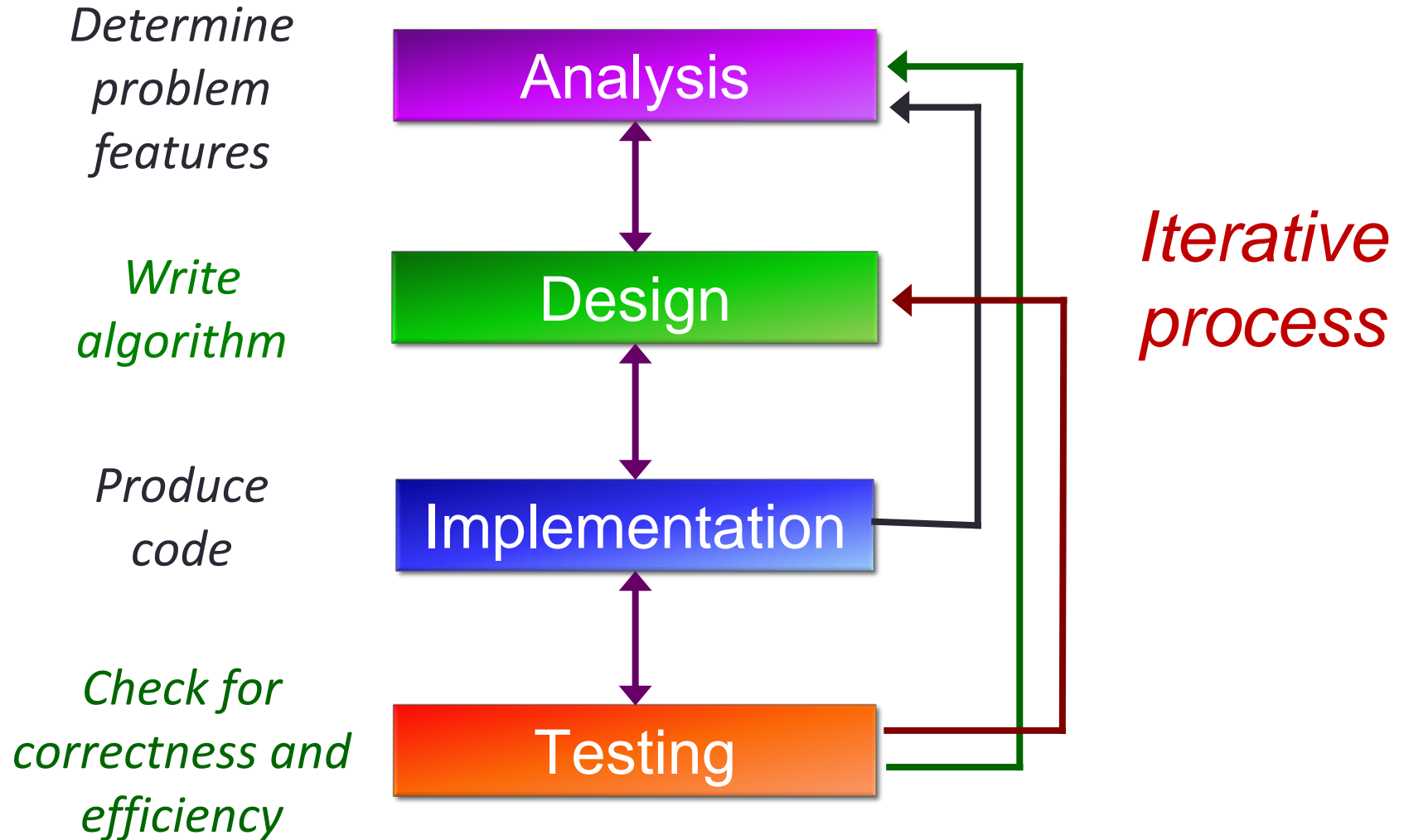
# Unit 4: Top-Down Design & Functions (1/2)

1. Problem Solving
2. Case Study: Top-Down Design
  - Computing the weight of a batch of flat washers
  - Incremental Refinement (some hierarchical chart)
  - Top-down design (of program) with structure charts
3. Function Prototypes
4. Default Return Type
5. 'return' statement in main()

# Unit 4: Top-Down Design & Functions (2/2)

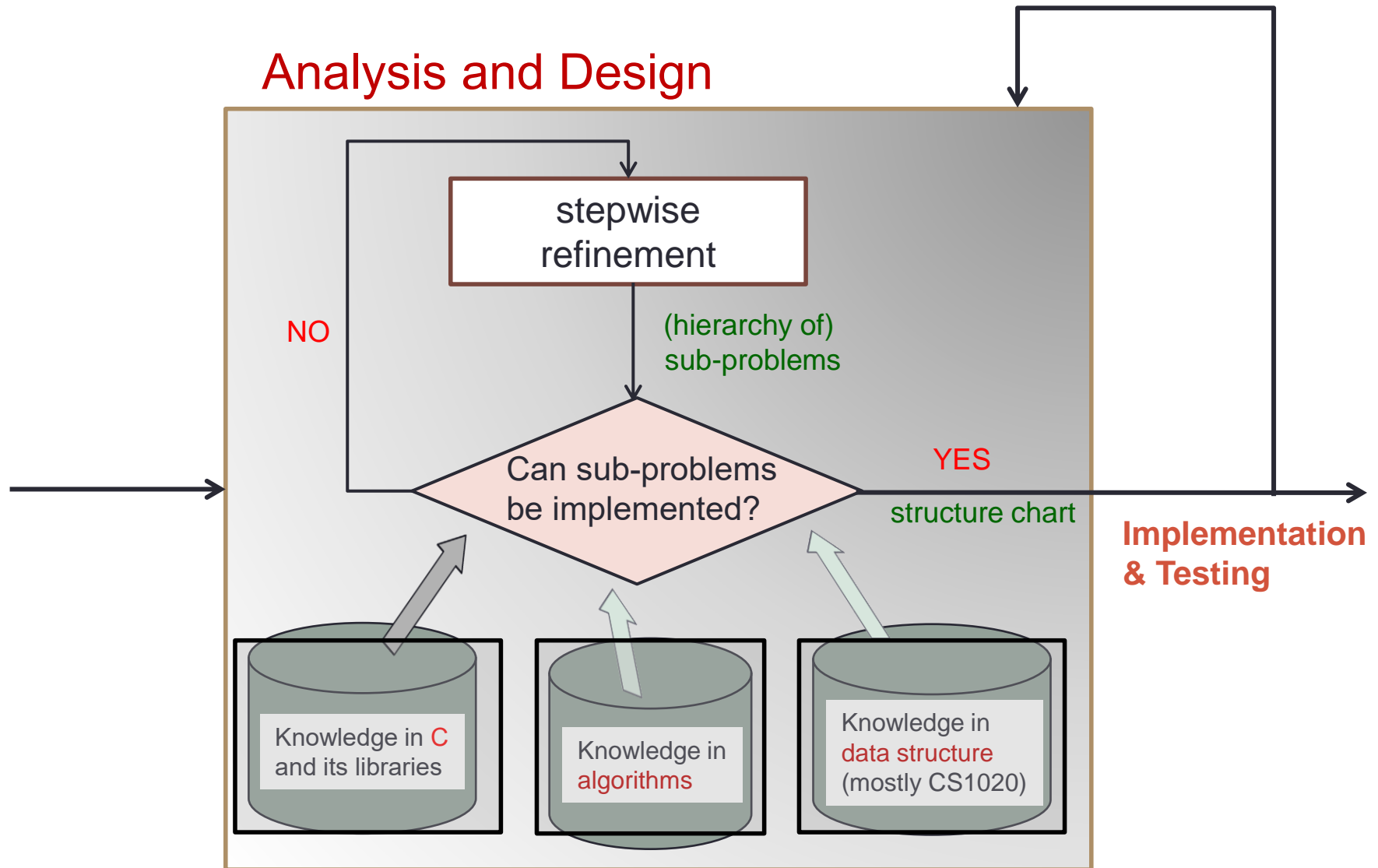
- 6. Writing Functions
- 7. Exercise #1: A Simple “Drawing” Program
- 8. Pass-By-Value and Scope Rules
- 9. Global Variables

# Problem Solving (1/2)

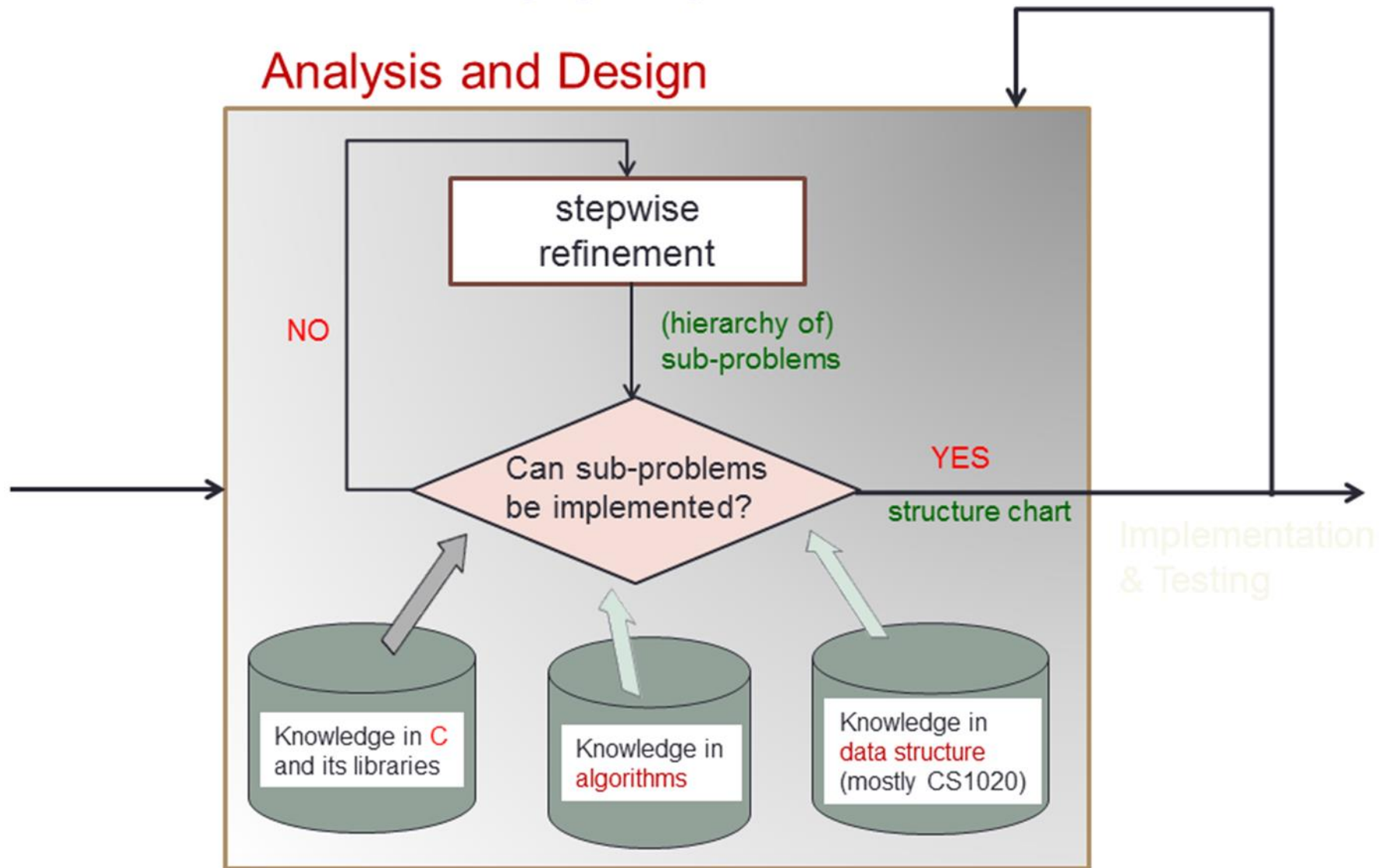




# Problem Solving (2/2)



# Problem Solving (2/2)

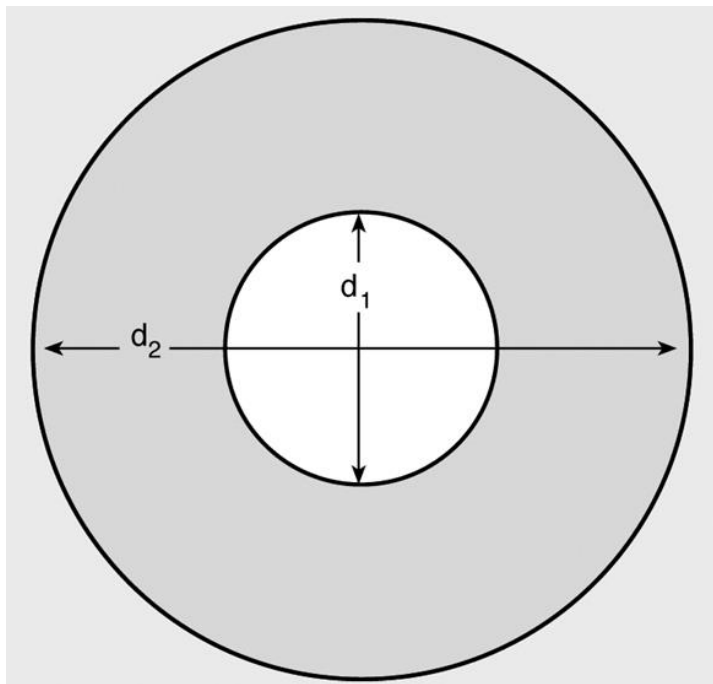


# Top-Down Design (1/13)

- We introduced some **math functions** in the previous unit.
- Such functions provide code reusability. Once the function is defined, we can use it whenever we need it, and as often as we need it.
- These math functions are provided in `<math.h>`. What if we want to define our own functions and use them?
- In the following case study, we introduce top-down design in approaching an algorithm problem.
- In the process, we encounter certain tasks that are similar, hence necessitating the creation of user-defined function.

# Top-Down Design (2/13)

**Case Study:** You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.

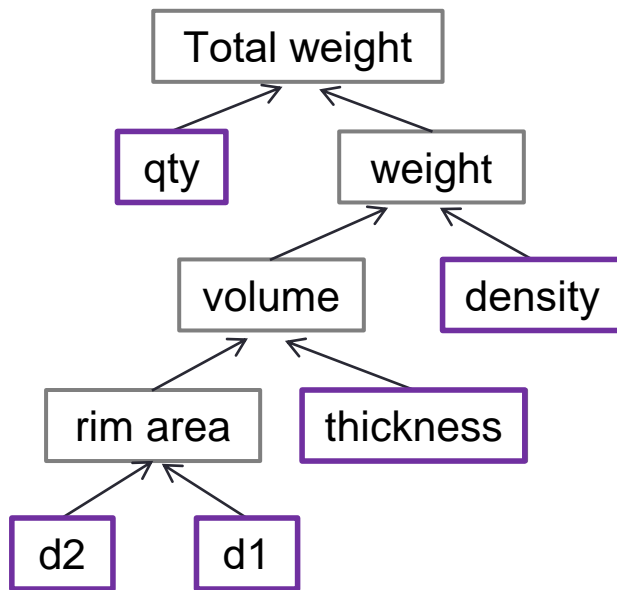


$$\text{rim area} = \pi(d_2/2)^2 - \pi(d_1/2)^2$$

# Top-Down Design (3/13)

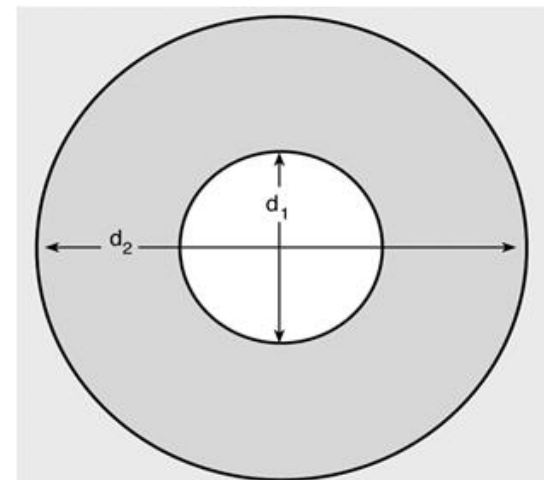
## ■ Analysis:

- To get the weight of a specified **qty** of washers, we need to know the **weight** of each washer
- To get the weight of a washer, we need its **volume** and **density** (weight = volume × density)
- To get the volume, we need its **rim area** and **thickness** (volume = rim area × thickness)
- To get the rim area, we need the diameters **d2** and **d1**



qty, density,  
thickness, d2  
and d1 are given  
as inputs

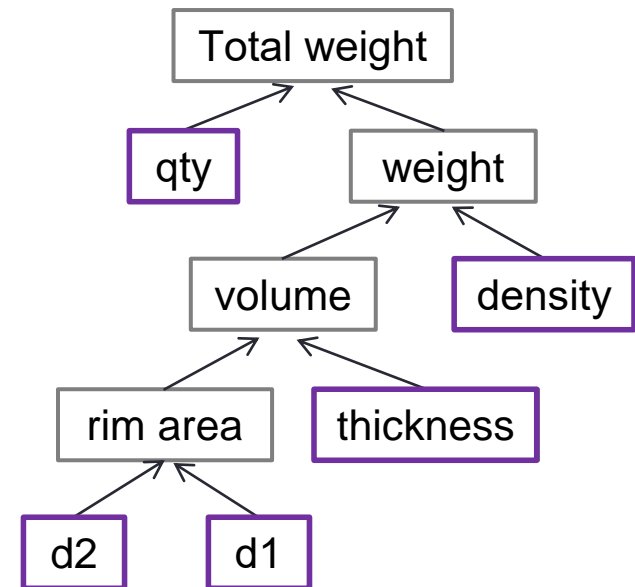
$$\text{rim area} = \pi(d_2/2)^2 - \pi(d_1/2)^2$$



# Top-Down Design (4/13)

## ■ Design (pseudocode):

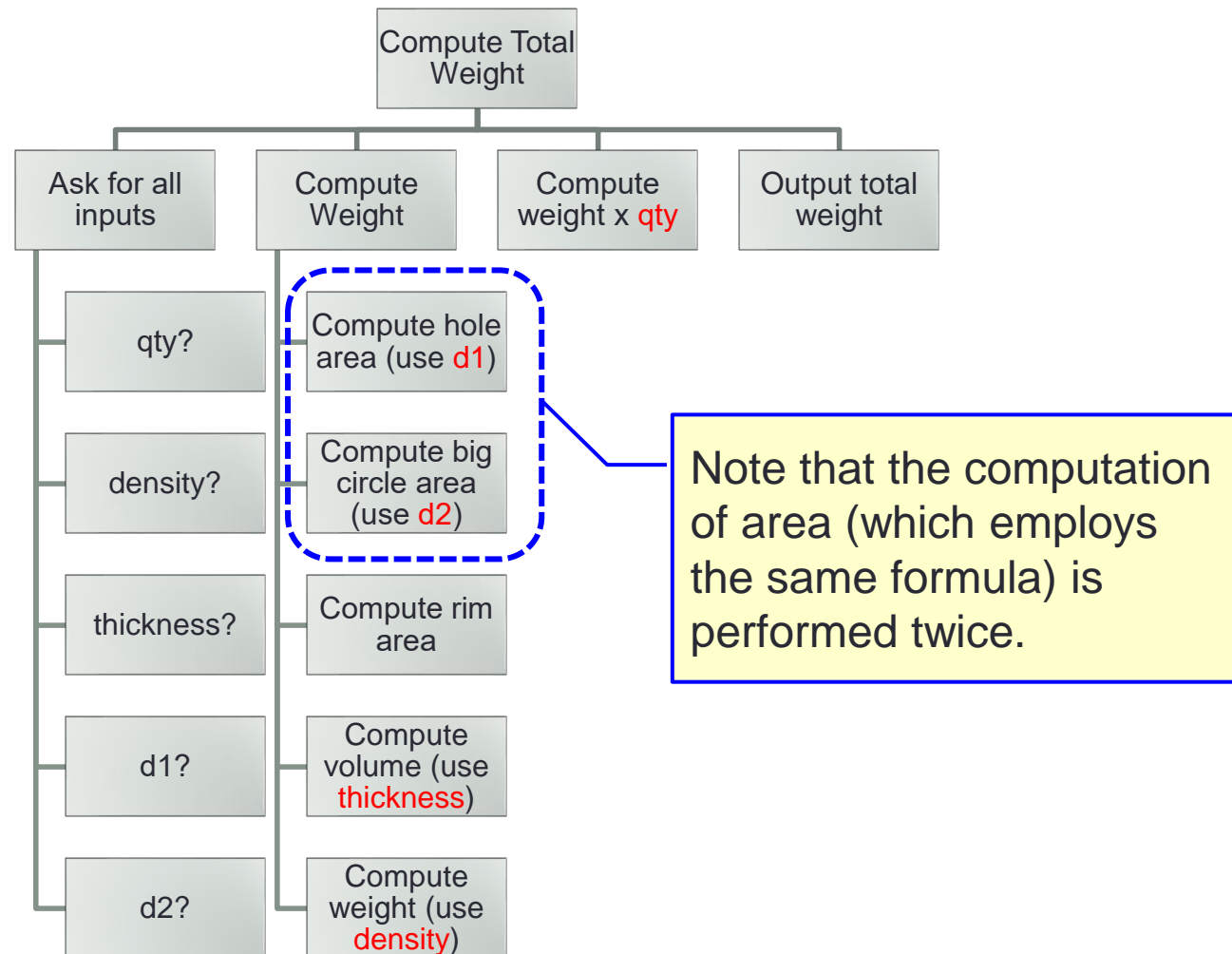
1. Read inputs (*qty*, *density*, *thickness*, *d2*, *d1*)
2. Compute weight of one washer
  - 2.1 Compute **area of small circle** (hole) using *d1*
  - 2.2 Compute **area of big circle** using *d2*
  - 2.3 Subtract small area from big area to get **rim area**
  - 2.4 Compute **volume** = rim area × thickness
  - 2.5 Compute **weight** = volume × density
3. Compute **total weight** of specified number of washer = weight × qty
4. Output the calculated total weight



**Step-wise refinement:** Splitting a complex task (step 2) into subtasks (steps 2.1 – 2.5)

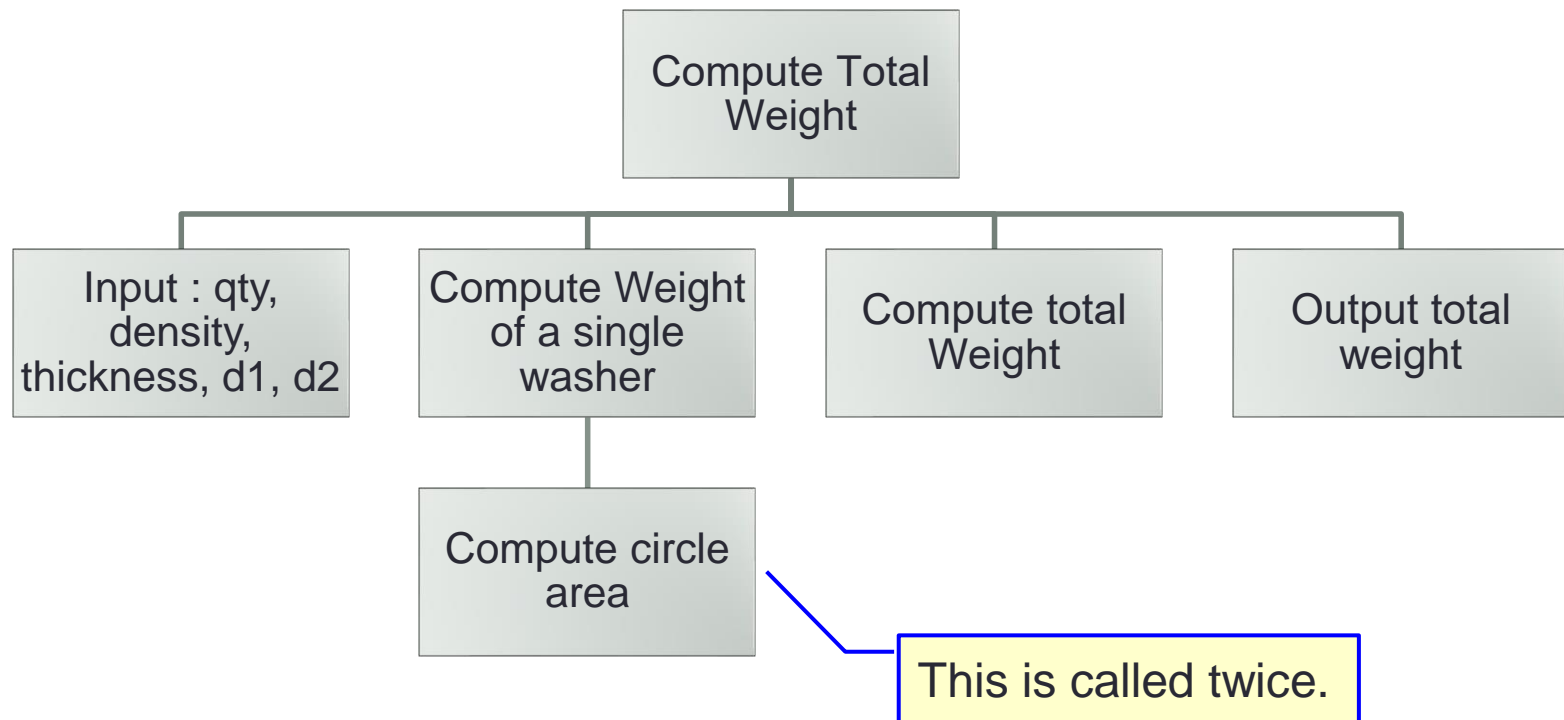
# Top-Down Design (5/13)

- Design (hierarchical chart):



# Top-Down Design (6/13)

- Design (structure chart):
  - A documentation tool that shows the **relationship** among the sub-tasks





# Top-Down Design (7/13)

Unit4\_Washers.c

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159

int main(void) {
    double d1,           // hole circle diameter
           d2,           // big circle diameter
           thickness,
           density;

    int    qty;

    double unit_weight, // single washer's weight
           total_weight, // a batch of washers' total weight
           outer_area,   // area of big circle
           inner_area,   // area of small circle
           rim_area;     // single washer's rim area

    // read input data
    printf("Inner diameter in cm: "); scanf("%lf", &d1);
    printf("Outer diameter in cm: "); scanf("%lf", &d2);
    printf("Thickness in cm: "); scanf("%lf", &thickness);
    printf("Density in grams per cubic cm: "); scanf("%lf", &density);
    printf("Quantity: "); scanf("%d", &qty);
```

# Top-Down Design (8/13)

Unit4\_Washers.c

```
// compute weight of a single washer
outer_area = pow(d2/2, 2) * PI;
inner_area = pow(d1/2, 2) * PI;
rim_area = outer_area - inner_area;
unit_weight = rim_area * thickness * density;

// compute weight of a batch of washers
total_weight = unit_weight * qty;

// output
printf("Total weight of the batch of %d washers is %.2f grams.\n",
      qty, total_weight);

return 0;
}
```

gcc -Wall Unit4\_washers.c -lm

# Top-Down Design (9/13)

- Note that area of circle is computed twice. For code reusability, it is better to define a function to compute area of a circle.

```
double circle_area(double diameter) {  
    return pow(diameter/2, 2) * PI;  
}
```

- We can then call/invoke this function whenever we need it.

`circle_area(d2)` → to compute area of circle with diameter `d2`

`circle_area(d1)` → to compute area of circle with diameter `d1`

# Top-Down Design (10/13)

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159
```

```
double circle_area(double diameter) {
    return pow(diameter/2, 2) * PI;
}
```

Function  
definition

```
int main(void) {
```

```
    // identical portion omitted for brevity
```

```
    // compute weight of a single washer
```

```
    rim_area = circle_area(d2) - circle_area(d1);
```

```
    unit_weight = rim_area * thickness * density;
```

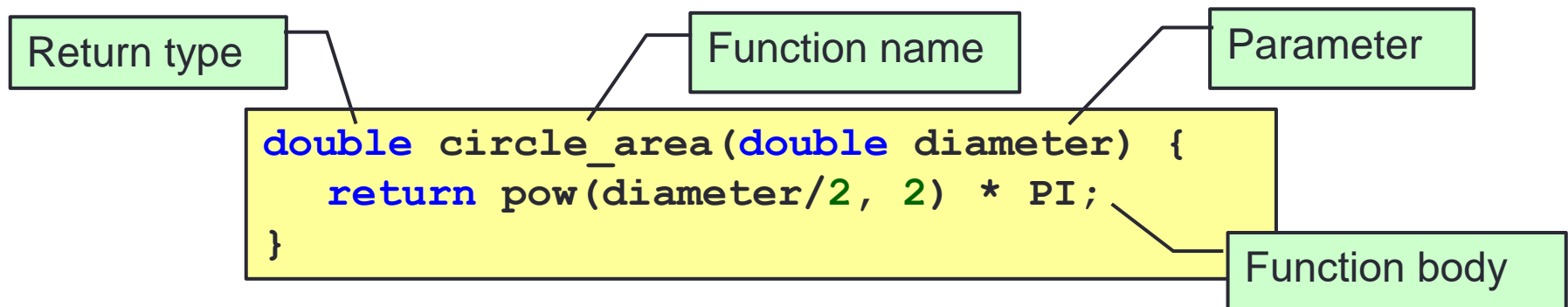
```
    // identical portion omitted for brevity
```

```
}
```

Calling `circle_area()`  
twice.

# Top-Down Design (11/13)

- Components of a **function definition**
  - Header (or signature): consists of **return type**, **function name**, and **a list of parameters** (with their types) separated by commas
  - Function names follow identifier rules (just like variable names)
    - May consist of letters, digit characters, or underscore, but cannot begin with a digit character
  - Return type is **void** if function does not need to return any value
  - Function body: code to perform the task; contains a **return** statement if return type is not void



# Top-Down Design (12/13)

```
double circle_area(double diameter)
{
    return pow(diameter/2, 2) * PI;
}
```

- Values of arguments are copied into parameters

```
rim_area = circle_area(d2) - circle_area(d1);
```

Value of **d2** copied to  
parameter **diameter**

Value of **d1** copied to  
parameter **diameter**

- Arguments need not be variable names; they can be constant values or expressions

`circle_area(12.3)` → To compute area of circle with diameter 12.3

`circle_area((a+b)/2)` → To compute area of circle with diameter (a+b)/2, where a and b are variables

# Top-Down Design (13/13)

- Preferred practice: add **function prototype**
  - Before main() function
  - Parameter names may be omitted, but not their type

Unit4\_WashersV2.c

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159
```

```
double circle_area(double);
```

Function prototype

```
int main(void) {
    // identical portion omitted for brevity

    // compute weight of a single washer
    rim_area = circle_area(d2) - circle_area(d1);
    unit_weight = rim_area * thickness * density;

    // identical portion omitted for brevity
}
```

Line 32 (see slide 21)

Line 45 (see slide 21)

```
double circle_area(double diameter) {
    return pow(diameter/2, 2) * PI;
}
```

Function definition

# Function Prototypes (1/2)

- It is a good practice to put **function prototypes** at the top of the program, before the `main()` function, to inform the compiler of the functions that your program may use and their return types and parameter types.
- Function definitions to follow after the `main()` function.
- Without function prototypes, you will get error/warning messages from the compiler.



## Function Prototypes (2/2)

- Let's remove (or comment off) the function prototype for `circle_area()` in `Unit4_WashersV2.c`
- Messages from compiler:

```
Unit4_WashersV2.c: In function 'main':
Unit4_WashersV2.c:32:5: warning: implicit declaration of
function 'circle_area'
Unit4_WashersV2.c: At top level:
Unit4_WashersV2.c:45:8: error: conflicting types for
'circle-area'
Unit4_WashersV2.c:32:16: previous implicit declaration of
'circle_area' was here
```

- Without function prototype, compiler assumes the default (implicit) return type of `int` for `circle_area()` when the function is used in line 32, which conflicts with the function header of `circle_area()` when the compiler encounters the function definition later in line 45.

# Default Return Type (1/3)

- A 'type-less' function has default return type of `int`

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("%d\n", f(100, 7));
5     return 0;
6 }
7
8 f(int a, int b) {
9     return a*b*b;
10 }
```

- Program can be compiled, but with warning:

```
warning: implicit declaration of function 'f' ← line 4
(due to absence of function prototype)
warning: return type defaults to 'int' ← line 8
```

# Default Return Type (2/3)

- Another example

```
1 #include <stdio.h>
2
3 int main(void) {
4     f(100, 7);
5     return 0;
6 }
7
8 void f(int a, int b) {
9     return a*b*b;
10 }
```

Without function prototype, compiler assumes function **f** to be an **int** function when it encounters this.

However, **f** is defined as a void function here, so it conflicts with above.

- Program can be compiled, but with warning:

```
warning: implicit declaration of function 'f' ← line 4
(due to absence of function prototype)
warning: conflicting types for 'f' ← line 8
```

# Default Return Type (3/3)

- **Tips**
  - Provide function prototypes for all functions
  - Explicitly specify the function return type for all functions

```
1 #include <stdio.h>
2
3 [int f(int, int);]
4
5 int main(void) {
6     printf("%d\n", f(100, 7));
7     return 0;
8 }
9
10 int f(int a, int b) {
11     return a*b*b;
12 }
```

# 'return' statement in main()

- Q: Why do we write `return 0;` in our `main()` function?
- Answer:
  - Our `main()` function has this header  
`int main(void)`
  - Hence it must return an integer (to the operating system)
  - The value 0 is chosen to be returned to the operating system (which is UNIX in our case). This is called the `status code` of the program.
  - In UNIX, when a program terminates with a status code of 0, it means a successful run.
  - You may optionally check the status code to determine the appropriate follow-up action. In UNIX, this is done by typing `echo $?` immediately after you have run your program. – You do not need to worry about this.

# Writing Functions (1/5)

- A **program** is a collection of functions (modules) to transform inputs to outputs
- In general, **each box** in a structure chart is a sub-problem which is handled by a **function**
- In mathematics, a **function** maps some input values to a **single** (possibly multiple dimensions) output
- In C, a **function** maps some input values to **zero or more** output values
  - **No output:** `void func(...) { ... }`
  - **One output**, e.g., `double func(...) { ...; return value; }`
  - **More outputs** through **changing input values** (we'll cover this later)
- **Return value** (if any) from function call can (but need not) be assigned to a variable.

# Writing Functions (2/5)

## Syntax:

```
function interface comment  
ftype fname (formal parameters list)  
{  
    local variable declarations  
    executable statements  
    return statement (if appropriate)  
}
```

### Unit4\_FunctionEg.c

```
/*  
 * Finds the square root of the  
 * sum of the squares of the two parameters  
 * Precond: x and y are non-negative numbers  
 */  
double sqrt_sum_square(double x, double y) {  
    // x and y above are the formal parameters  
  
    double sum_square; // local variable declaration  
  
    sum_square = pow(x,2) + pow(y,2);  
    return sqrt(sum_square);  
}
```

## Notes:

**Precondition:** describes conditions that should be true **before calling** function.

**Postcondition:** describes conditions that should be true **after executing** function.

These are for documentation purpose.

# Writing Functions (3/5)

Actual parameters (also arguments) are values passed to function for computation

Formal parameters (or simply parameters) are placeholder when function is defined.

- Matching of actual and formal parameters from left to right
- Scope of formal parameters, local variables are within the function only

```
// Function prototype at top of program
double sqrt_sum_square(double, double);

int main(void) {
    double y = 1.23; // not the same as y
                    // in sqrt_sum_square

    double z = 4.56;

    // x below not the same as x in sqrt_sum_square
    double x = sqrt_sum_square(y, z);

    printf("The square root of the sum of square ");
    printf("of %.2f and %.2f is %.2f\n", y, z, x);
    return 0;
}
```

```
double sqrt_sum_square(double x, double y)
{
    // x and y above are formal parameters

    double sum_square; // local variable

    sum_square = pow(x,2) + pow(y,2);

    return sqrt(sum_square);
}
```

- Arrows indicate flow of control between main() and the function
- Add function prototype at top of program, before main() function



# Writing Functions (4/5)

The complete program

Unit4\_FunctionEg.c

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

/* Function prototype placed at top of program */
double sqrt_sum_square(double, double);

int main(void) {
    double y = 1.23;    // not the same as y in sqrt_sum_square
    double z = 4.56;

    // x below has nothing to do with x in sqrt_sum_square
    double x = sqrt_sum_square( y, z );
    // in the previous statement, y and z are actual parameters

    printf("The square root of the sum of squares ");
    printf("of %.2f and %.2f is %.2f\n", y, z, x);

    return 0;
}

/* Finds the square root of the
 * sum of the squares of the two parameters
 * Precond: x and y are non-negative numbers
 */
double sqrt_sum_square(double x, double y) {
    // x and y above are the formal parameters

    double sum_square; // local variable declaration

    sum_square = pow(x,2) + pow(y,2);
    return sqrt(sum_square);
}
```

# Writing Functions (5/5)

- Use of functions allow us to manage a complex (abstract) task with a number of simple (specific) ones.
  - This allows us to switch between **abstract** and go to **specific** at ease to eventually solve the problem.
- Function allows a team of programmers **working together** on a large program – each programmer will be responsible for a particular set of functions.
- Function is good mechanism to allow **re-use** across different programs. Programmers use functions like **building blocks**.
- Function allows **incremental implementation** and **testing** (with the use of **driver** function to call the function and then to check the output)
- Acronym **NOT** summarizes the requirements for argument list correspondence. (**N**: number of arguments, **O**: order, and **T**: type).

# Ex #1: A Simple “Drawing” Program (1/3)

## Problem:

- Write a program `Unit4_DrawFigures.c` to draw a rocket ship (which is a triangle over a rectangle over an inverted V), a male stick figure (a circle over a rectangle over an inverted V), and a female stick figure (a circle over a triangle over an inverted V)



*rocket*



*male*



*female*

## Analysis:

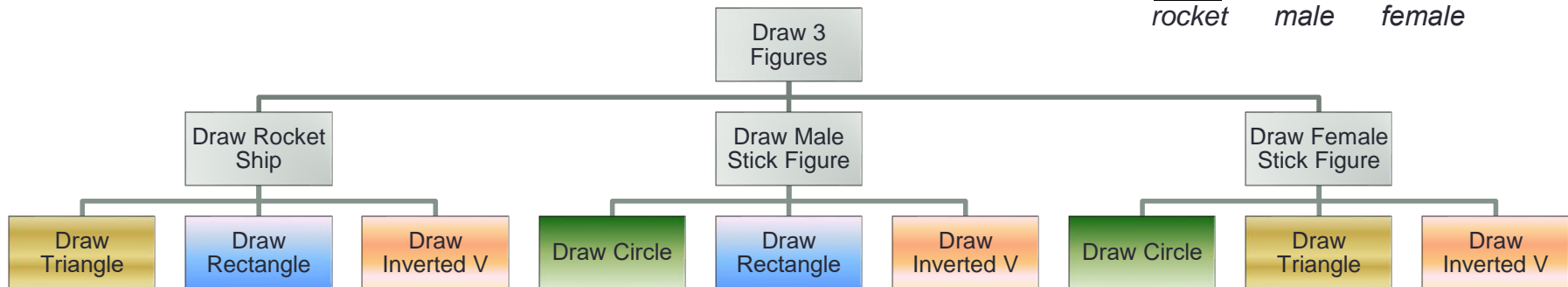
- No particular input needed, just draw the needed 3 figures
- There are common shapes shared by the 3 figures

## Design:

- Algorithm (in words):
  1. Draw Rocket ship
  2. Draw Male stick figure (below Rocket ship)
  3. Draw Female stick figure (below Male stick figure)

# Ex #1: A Simple “Drawing” Program (2/3)

Design (Structure Chart):



*rocket*



*male*



*female*

# Ex #1: A Simple “Drawing” Program (3/3)

Implementation (partial program)

Unit4\_DrawFiguresPartial.c

```
#include <stdio.h>

void draw_rocket_ship();
void draw_male_stick_figure();
void draw_circle();
void draw_rectangle();

int main(void) {
    draw_rocket_ship();
    printf("\n\n");

    draw_male_stick_figure();
    printf("\n\n");

    return 0;
}
```

```
void draw_rocket_ship() {
}

void draw_male_stick_figure() {
}

void draw_circle() {
    printf("  **  \n");
    printf(" *    * \n");
    printf(" *    * \n");
    printf("  **  \n");
}

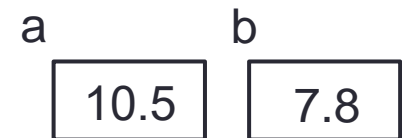
void draw_rectangle() {
    printf(" ***** \n");
    printf(" *      * \n");
    printf(" *      * \n");
    printf(" *      * \n");
    printf(" ***** \n");
}
```

Write a complete program  
Unit4\_DrawFigures.c

# Pass-By-Value and Scope Rules (1/4)

- In C, the actual parameters are passed to the formal parameters by a mechanism known as **pass-by-value**.

```
int main(void) {
    double a = 10.5, b = 7.8;
    → printf("%.2f\n", sqrt_sum_square(3.2, 12/5));
    → printf("%.2f\n", sqrt_sum_square(a, a+b));
    return 0;
}
```



Actual parameters:  
10.5 and 2003

```
double sqrt_sum_square(double x, double y) {
    double sum_square;
    sum_square = pow(x,2) + pow(y,2);
    return sqrt(sum_square);
}
```

Formal parameters:



# Pass-By-Value and Scope Rules (2/4)

- Formal parameters are local to the function they are declared in.
- Variables declared within the function are also local to the function.
- **Local parameters and variables are only accessible in the function they are declared** – scope rule.
- When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.
- Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is released.
- Hence, local parameters and variables of a function exist in memory only during the execution of the function. They are called **automatic variables**.
- In contrast, **static variables** exist in the memory even after the function is executed. (We will not use static variables in CS1010.)

# Pass-By-Value and Scope Rules (3/4)

- Spot the error in this code:

```
int f(int) ;  
  
int main(void) {  
    int a;  
    ...  
}  
  
int f(int x) {  
    return a + x;  
}
```

*Answer:*

Variable **a** is local to **main()**, not **f()**. Hence, variable **a** cannot be used in **f()**.



# Pass-By-Value and Scope Rules (4/4)

- Trace this code by hand and write out its output.

```
#include <stdio.h>
void g(int, int);

int main(void) {
    int a = 2, b = 3;
```

In main, before: a=2, b=3

In g, before: a=2, b=3

In g, after : a=102, b=203

In main, after : a=2, b=3

```
→ printf("In main, before: a=%d, b=%d\n", a, b);
   g(a, b);
→ printf("In main, after : a=%d, b=%d\n", a, b);
   return 0;
}

void g(int a, int b) {
→ printf("In g, before: a=%d, b=%d\n", a, b);
   a = 100 + a;
   b = 200 + b;
→ printf("In g, after : a=%d, b=%d\n", a, b);
}
```

# Global Variables (1/2)

- Global variables are those that are declared outside all functions.

```
int f1(int);  
void f2(double);  
  
int glob; // global variable  
  
int main(void) {  
    ...  
    glob = glob + 1;  
}  
  
int f1(int x) {  
    ...  
    glob = glob + 1;  
}  
  
void f2(double x) {  
    ...  
    glob = glob + 1;  
}
```

## Global Variables (2/2)

- Global variables can be accessed and modified by any function!
- Because of this, it is hard to trace when and where the global variables are modified.
- Hence, we will NOT allow the use of global variables

# Summary

- In this unit, you have learned about
  - Top-down design through stepwise refinement, splitting a task into smaller sub-tasks
  - How to write user-defined functions and use them
  - Pass-by-value and scope rules of local parameters and variables

# End of File