CS1010 http://www.comp.nus.edu.sg/~cs1010/

Programming Methodology

UNIT 5

Top-Down Design & Functions

NUS National University of Singapore | School of Computing

Unit 5: 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 3 Top-Down Design with Functions

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

Math Functions (1/2)

- In C, there are many libraries offering functions for you to use.
- Eg: scanf() and printf() – requires to include <stdio.h>
- In Exercise #5, for t^2 you may use **`t*t`**, or the pow() function in the math library: **`pow(t, 2)`**
  - **`pow(x, y) //computes x raised to the power of y`**
- To use math functions, you need to
  - Include **`<math.h>`** AND
  - Compile your program with **`-lm`** option (i.e. **`gcc -lm`** …)
- See Tables 3.3 and 3.4 (pages 88 – 89) for some math functions

Math Functions (2/2)

- Some useful math functions
  - Function abs(x) from <stdlib.h>; the rest from <math.h>

Function	Arguments	Result
abs(x)	int	int
ceil(x)	double	double
cos(x)	double (radians)	double
exp(x)	double	double
fabs(x)	double	double
floor(x)	double	double
log(x)	double	double
log10(x)	double	double
ceil(x)	double	double
pow(x, y)	double, double	double
sin(x)	double (radians)	double
sqrt(x)	double	double
tan(x)	double (radians)	double

Function prototype:

double pow(double x, double y)

function return type

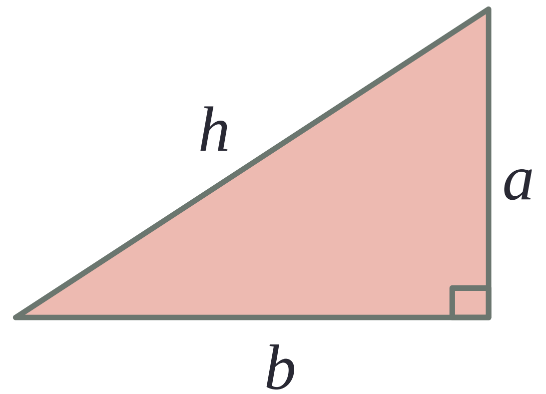
Q: Since the parameters x and y in pow() function are of double type, why can we call the function with pow(t, 2)?

A: Integer value can be assigned to a double variable/parameter.

Math Functions: Example (1/2)

- Program Unit3_Hypotenuse.c computes the hypotenuse of a right-angled triangle given the lengths of its two perpendicular sides

$$h = \sqrt{(a^2 + b^2)}$$



Math Functions: Example (2/2)

Unit3_Hypotenuse.c

```
// Unit3_Hypotenuse.c
// Compute the hypotenuse of a right-angled triangle.
#include <stdio.h>
#include <math.h>

int main(void) {
   float hypot, side1, side2;

   printf("Enter lengths of the 2 perpendicular sides: ");
   scanf("%f %f", &side1, &side2);

   hypot = sqrt(side1*side1 + side2*side2);
   // or hypot = sqrt(pow(side1, 2) + pow(side2, 2));

   printf("Hypotenuse = %6.2f\n", hypot);

   return 0;
}
```

Remember to compile with –lm option!

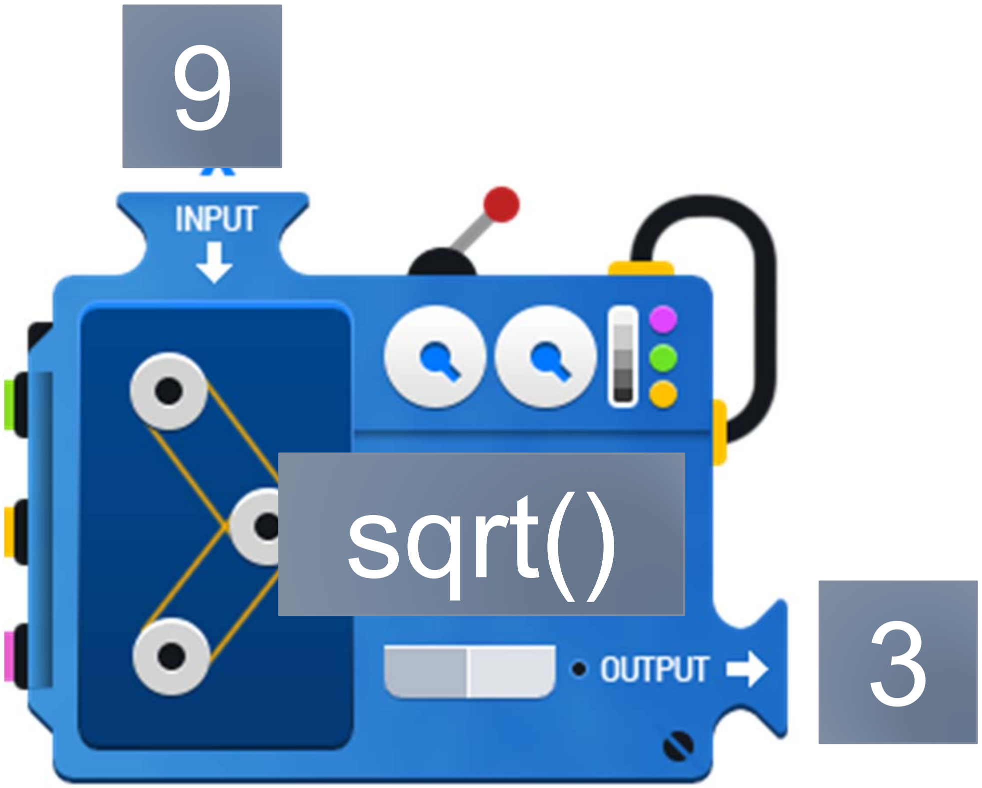
Math Functions: Example (2/2)

Unit3_Hypotenuse.c

```
// Unit3_Hypotenuse.c
// Compute the hypotenuse of a right-angled triangle.
#include <stdio.h>
#include <math.h>

int main(void) {
  float hypot, side1, side2;

  printf("Enter lengths of the 2 perpendicular sides: ");
  scanf("%f %f", &side1, &side2);

  hypot = sqrt(side1*side1 + side2*side2);
  // or hypot = sqrt(pow(side1, 2) + pow(side2, 2));

  printf("Hypotenuse = %6.2f\n", hypot);

  return 0;
}
```

Function

- "`sqrt()`" is a function
  - Input 9
  - Output 3



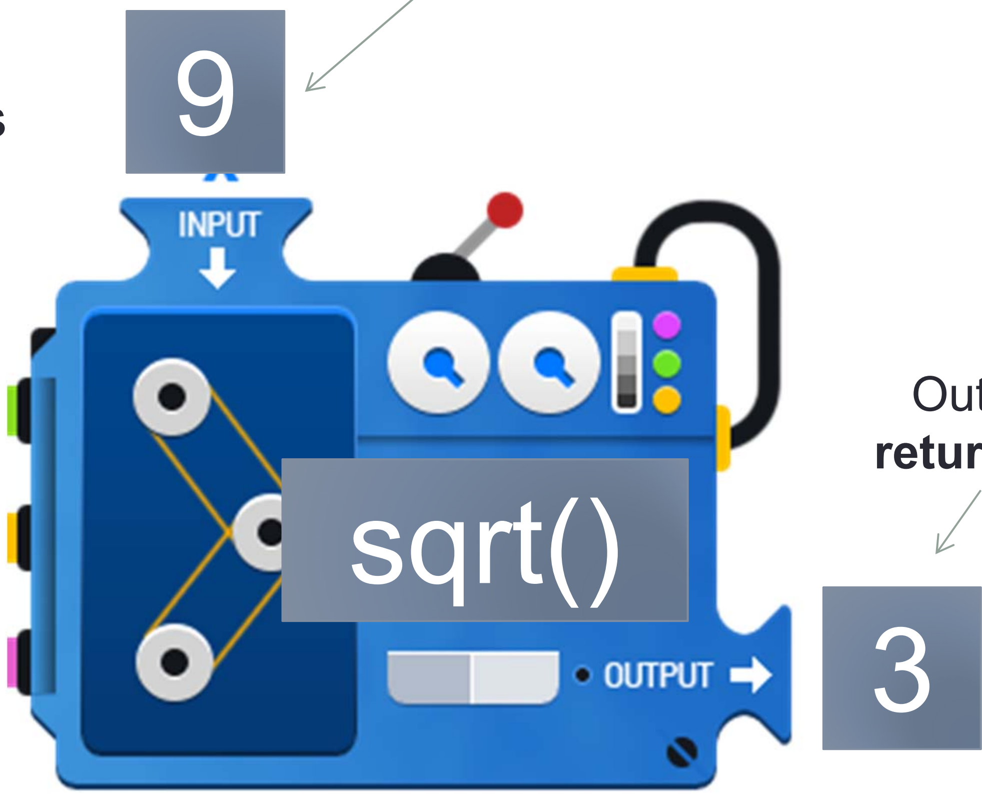
Writing Functions (1/5)

- 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.

Function

- In general, a function can have
  - **no** input, or
  - **no** output,
  - or any combinations

Input, or called **arguments**, or **parameters**



Output or **return value**

Writing Functions (2/5)

Syntax:

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

Unit5_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

Unit5_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).

WHY FUNCTIONS?

Writing Functions (4/5)

The complete program

Unit5_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);
}
```

Isn't it better?

```
#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(pow(y,2) + pow (z,2));
   // 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;
}
```

Is it always shorter the better?

- In Chinese we say:
  - “You learn and revise knowledge according to a regular schedule”
- In Chinese modern translation:
  - “學習知識并按一定的時間去溫習它”
- In Ancient Chinese
  - “學而時習之”

Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- You won't write code like

```
int binoCoff(int n, int k) {
    a = code for computing 1x2x3…x n
    b = code for computing 1x2x3…x k
    c = code for computing 1x2x3…x(n-k)
    return a / (b * c);
}
```

Common Patterns

Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- Write a function `factorial(n)` then

```
int binoCoff(int n, int k) {
    a = factorial(n)
    b = factorial(k)
    c = factorial(n-k)
    return a / (b * c);
}
```

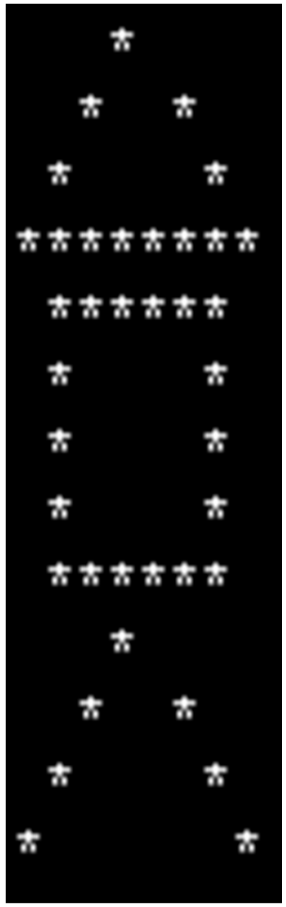
Advantages of Functions Abstraction

- Captures common patterns
- Makes it more natural to think about tasks and subtasks
- Makes programs easier to understand
- Allows for code reuse
  - E.g. the factorial function can be used for other functions, e.g. computing the natural number e
- Separates specification from implementation
  - I can ask my colleagues/project partners to write the function factorial for me
  - And he can write in any method he wants
- Make debugging easier
- Hide irrelevant information

Made of?

How doe

rk?



rocket

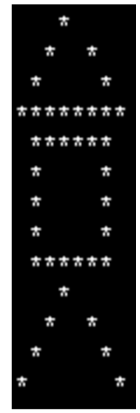
male

female

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

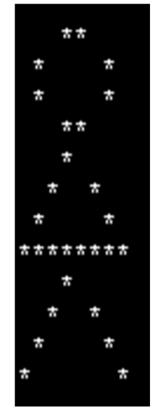
Problem:

- Write a program Unit5_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:

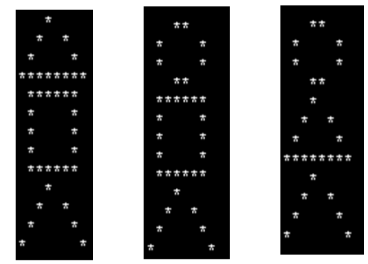
- No particular input needed, just draw the needed 3 figures
- There are common shapes shared by the 3 figures

Design:

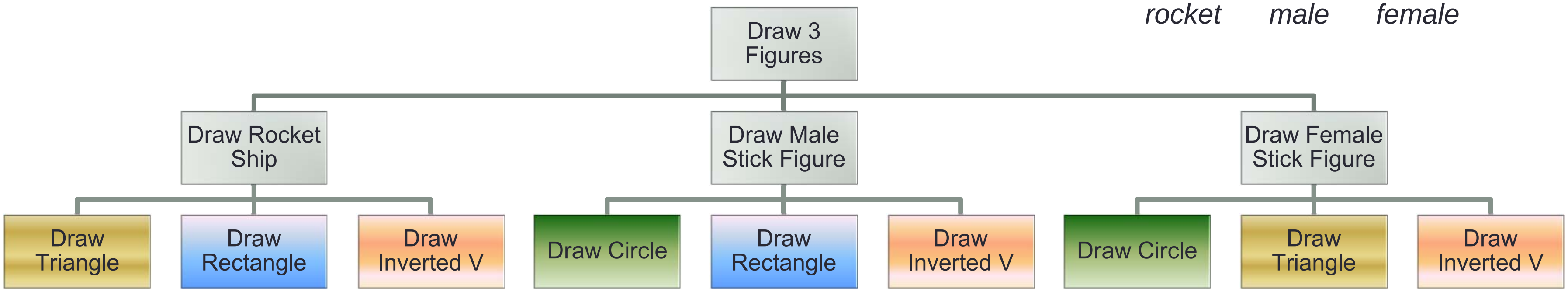
- Algorithm (in words):
  1. Draw Rocket ship
  2. Draw Male stick figure (below Rocket ship)
  3. Draw Female stick figure (below Male stick figure)

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)

Unit5_DrawFiguresPartial.c

```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;
}
```

```c
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 Unit5_DrawFigures.c

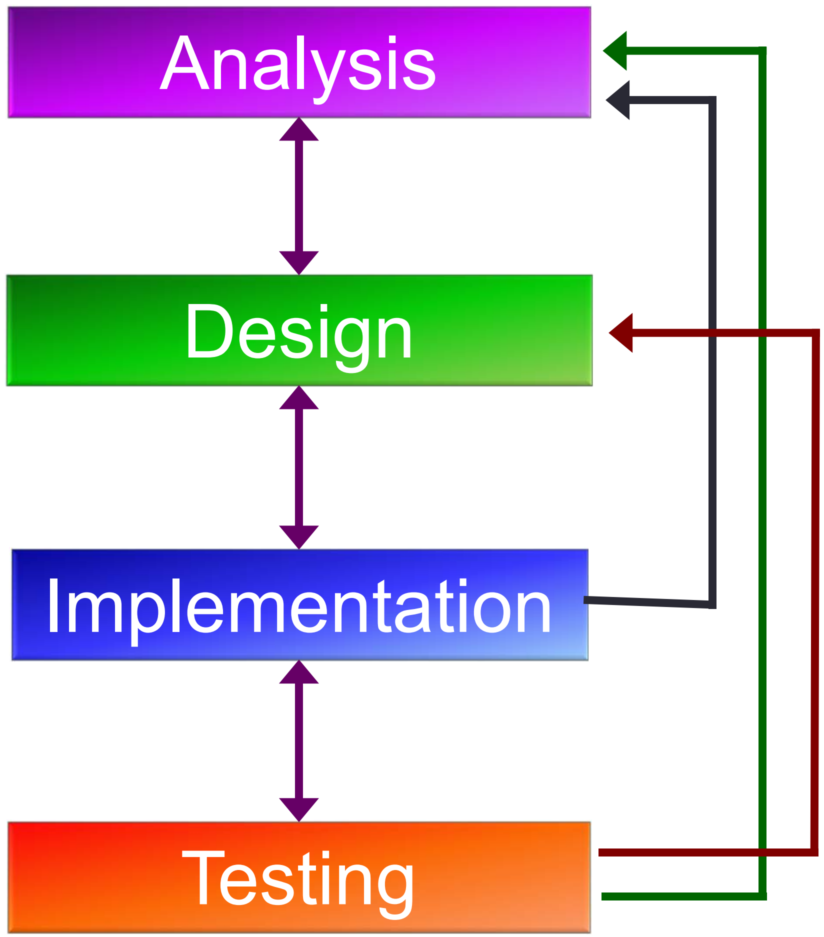
Problem Solving (1/2)

Determine problem features

Write algorithm

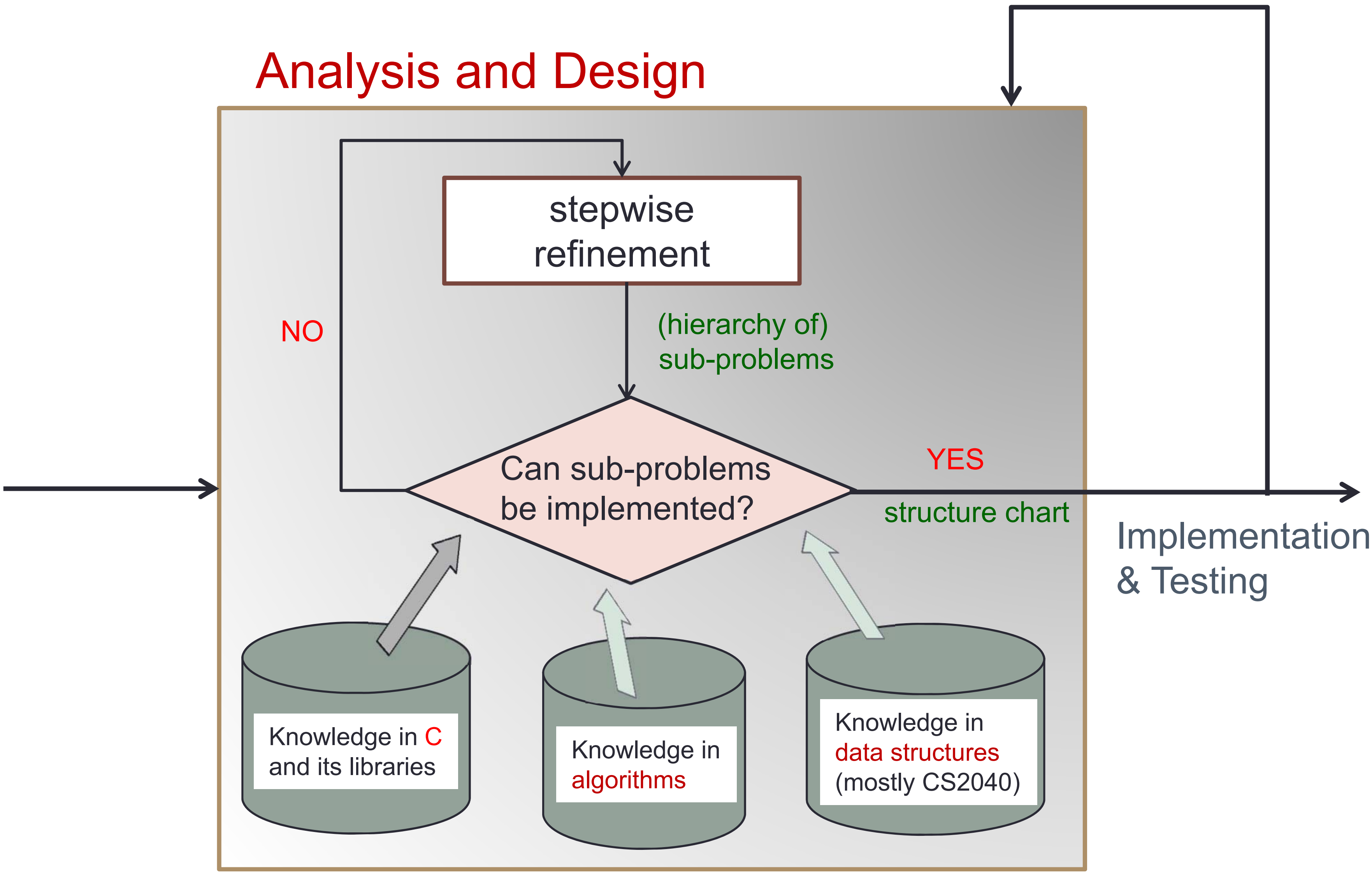
Produce code

Check for correctness and efficiency



Iterative process

Problem Solving (2/2)



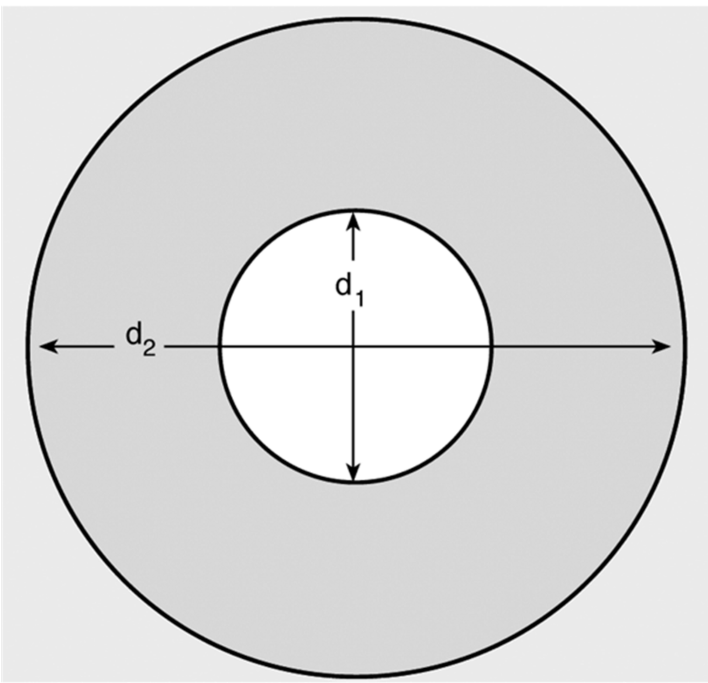
Top-Down Design (1/13)

- We introduced some math functions in the previous unit.
- These math functions are provided in <math.h>.
- 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.
- Can we create our own functions?
- 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.

Decomposition!

Top-Down Design (2/13)

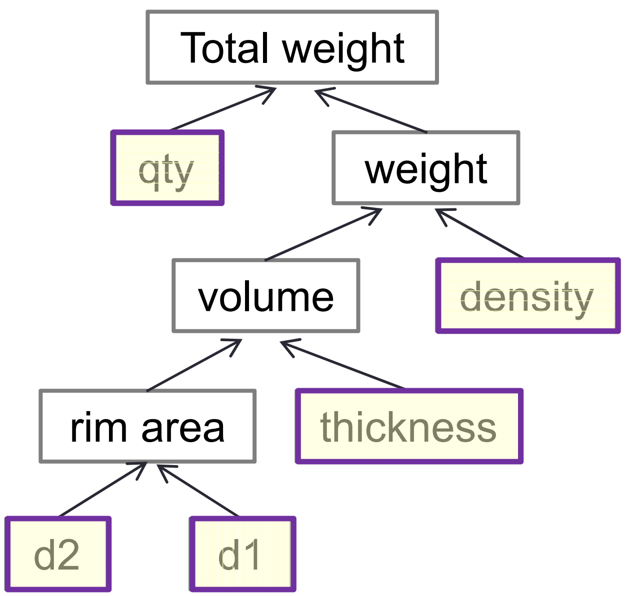
Case Study: You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.



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

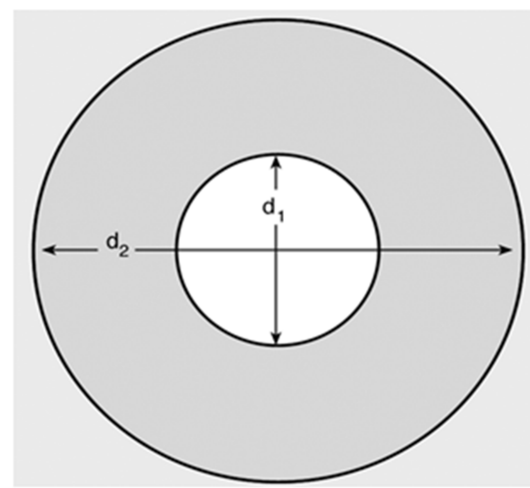
Top-Down Design (3/13)

- **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



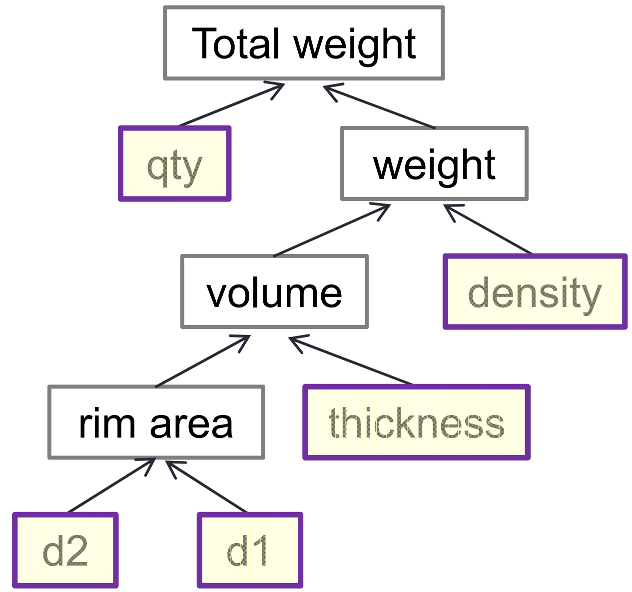
qty, density, thickness, d2 and d1 are given as inputs

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



Top-Down Design (4/13)

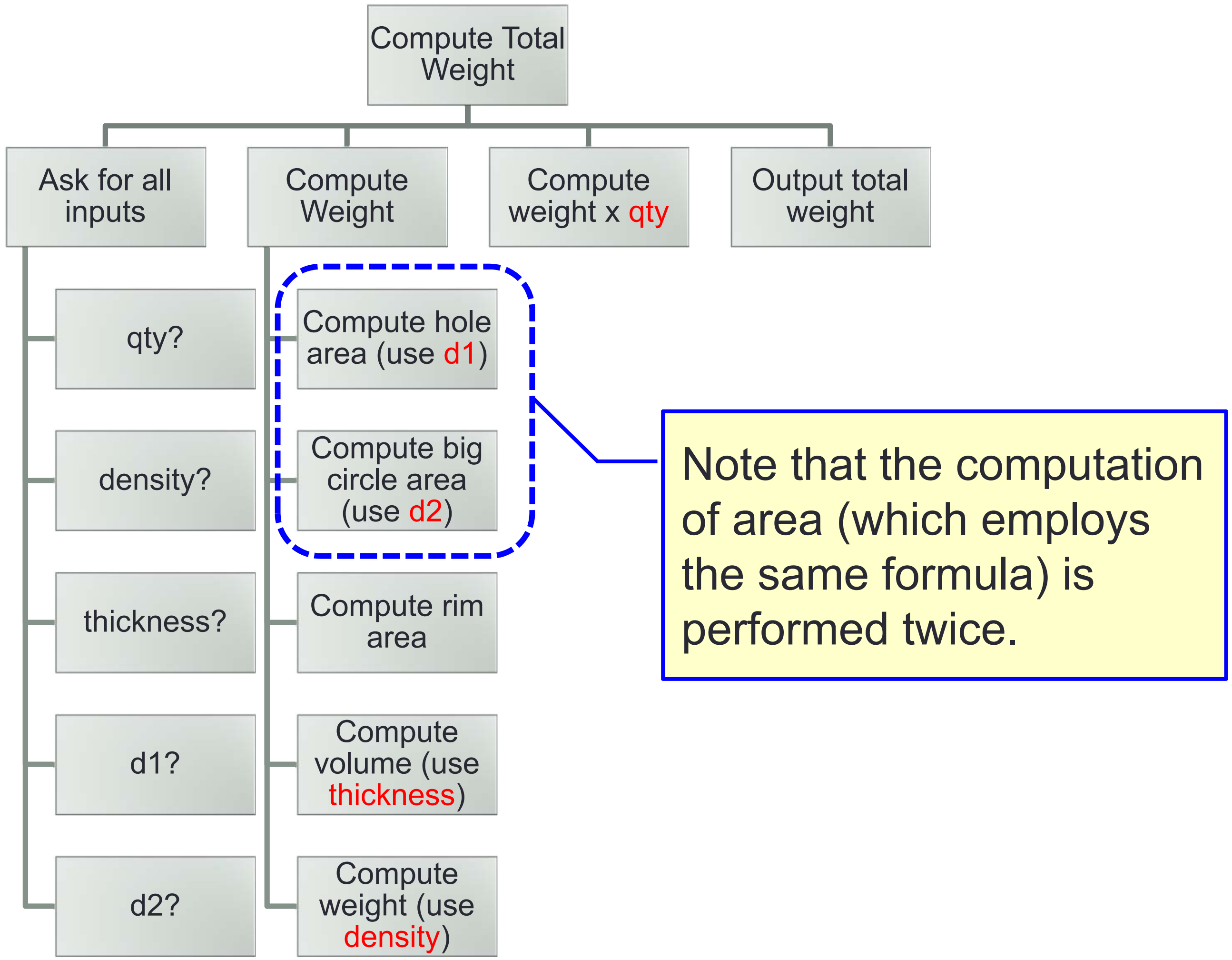
- Design (pseudocode):
  1. Read inputs (qty, density, thickness, d2, d1)
  2. Compute weight of one washer
     - 2.1 Compute area of small circle (hole) using d1
     - 2.2 Compute area of big circle using d2
     - 2.3 Subtract small area from big area to get rim area
     - 2.4 Compute volume = rim 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)

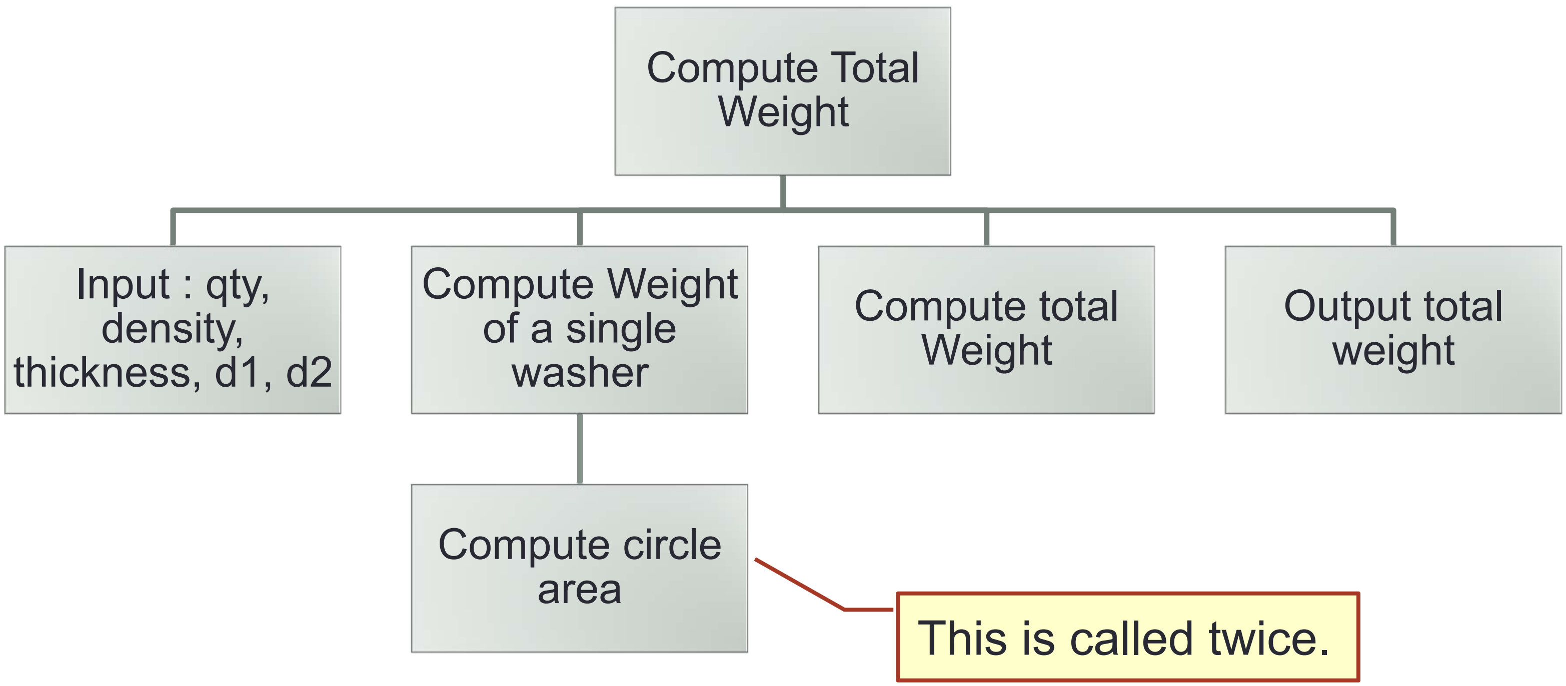
Top-Down Design (5/13)

- 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)

Unit5_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)

Unit5_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 Unit5_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;
}

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
}
```

Function definition

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 <u>cannot</u> 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

Return type

Function name

Parameter

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

Function body

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

Unit5_WashersV2.c

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

double circle_area(double);

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
}

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

Function prototype

Line 32 (see slide 21)

Line 45 (see slide 21)

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 Unit5_WashersV2.c
- Messages from compiler:

```
Unit5_WashersV2.c: In function 'main':
Unit5_WashersV2.c:32:5: warning: implicit declaration of
function 'circle_area'
Unit5_WashersV2.c: At top level:
Unit5_WashersV2.c:45:8: error: conflicting types for
'circle-area'
Unit5_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 <u>all functions</u>
  - Explicitly specify the function <u>return type</u> for all functions

```
#include <stdio.h>

int f(int, int);

int main(void) {
   printf("%d\n", f(100, 7));
   return 0;
}

int f(int a, int b) {
   return a*b*b;
}
```

'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.

There Is a Saying

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", srqt_sum_square(3.2, 12/5);
  printf("%.2f\n", srqt_sum_square(a, a+b);
  return 0;
}
```

a: 10.5 b: 7.8

Actual parameters:

3.2 and 2.0 / 10.5 and 18.3

```
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:

x: 3.2 / 10.5 y: 2.0 / 18.3

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;
}
```

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;

  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);
}
```

Consequence of Pass-By-Value

- Can this code be used to swap the values in **a** and **b**?

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

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

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

void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Pass-by-Value

- Your passport is important to you, you do not want others to make any changes to it and ruin in
- If you want to apply for a job, and the company needs your passport
  - 99% of the time they just need your photocopy of your passport
  - Why?
  - They also do not want to lost/damage your passport

- Pass-by-value
  - Make a copy of your data to the function

So Vegas is a function!

In contrast (in the future)

- If you want to apply a VISA to a certain country, you need to submit the passport
  - Why?
  - They need to stamp or paste the VISA in your passport

- The passport is changed during the process.
  - There are ways to change the variable you passed (but not now)
  - Namely, pass-by-pointer(later) and pass-by-reference(C++)

Writing Pre-Condition

- The function triangle_area() computes the area of a right-angled triangle. The two parameters are the lengths of the two perpendicular sides.
- How should you write the pre-condition?

```
// Compute the area of a right-angled triangle.
// side1 and side2 are the lengths of the
// two perpendicular sides.
// Pre-cond:  side1 > 0, side2 > 0
double triangle_area(double side1, double side2) {
  return side1 * side2 / 2.0;
}
```

Function Cohesion

- Which of the two approaches is correct?

```
// Compute the area of a right-angled triangle.
// Pre-cond: side1 > 0, side2 > 0
double triangle_area(double side1, double side2) {
   return side1 * side2 / 2.0;
}
```

```
// Compute the area of a right-angled triangle.
// Pre-cond: side1 > 0, side2 > 0
void triangle_area(double side1, double side2) {
   printf("Area = %.2f\n", side1 * side2 / 2.0);
}
```

- In general, a function should perform either computation or I/O, not both. triangle_area() is to compute the area, so it should return the answer to the caller, which then decides whether to print the answer or use it for further computation in a bigger task.

Global Variables (1/2)

- Global variables are those that are declared <u>outside all functions</u>.

```
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