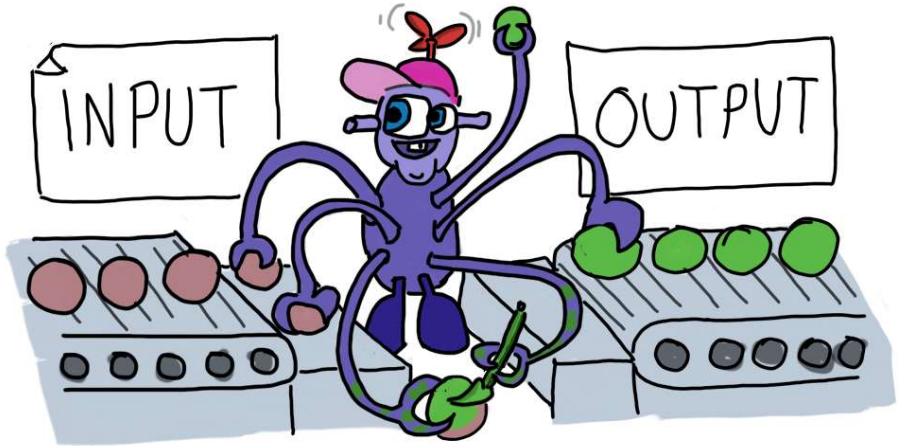


Top-Down Design with Functions

Mirza Mohammad Lutfe Elahi

Outline

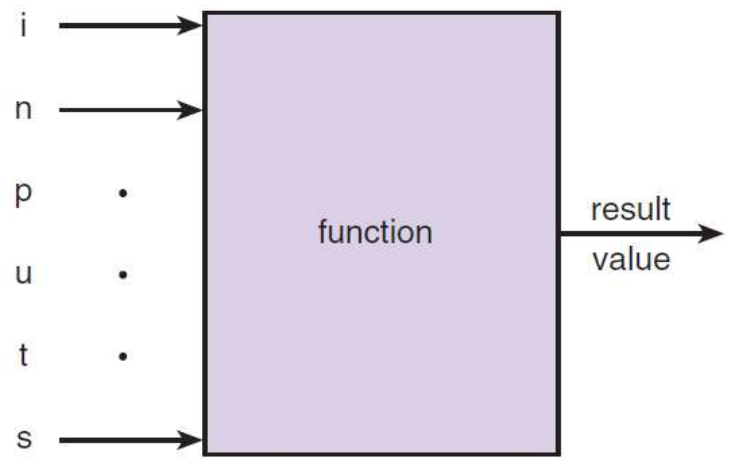
- Introduction to Functions
- Library Functions and Code Reuse
- Top-Down Design and Structure Charts
- Functions without Arguments or Results
- Function Prototypes, Definitions, and Function Calls
- Functions with Input Arguments and a Single Result
- Testing Functions and Function Data Area
- Advantages of Functions and Common Errors

Introduction to Functions

- So far, we know how to use operators: **+ - * / %** to form simple arithmetic expressions
- However, we are not yet able to write many other mathematical expressions we are used to
- For example, we cannot yet represent any of the following expressions in C: $\sqrt{x}$, e^x, $\log x$
- C does not have operators for *square root*, *exponential*, *log*, *sine*, etc.
- Instead, C provides program units called functions to carry out these and other mathematical operations

Introduction to Functions

- A function is like a box that receives input arguments and returns a result value
- For example: `y = sqrt(x);`
  - `x` is the input argument
  - The result value is assigned to `y`
- For example: `y = sqrt(16.0);`
  - The result value is `4.0`
- The math library provides many standard functions in C



Library Functions and Code Resue

- 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

Square Root Program

```
1.  /*
2.   * Performs three square root computations
3.   */
4.
5.  #include <stdio.h> /* definitions of printf, scanf */
6.  #include <math.h>  /* definition of sqrt           */
7.
8.  int
9.  main(void)
10. {
11.       double first, second,    /* input - two data values        */
12.              first_sqrt,       /* output - square root of first  */
13.              second_sqrt,      /* output - square root of second */
14.              sum_sqrt;         /* output - square root of sum    */
15.
16.       /* Get first number and display its square root. */
17.       printf("Enter the first number> ");
18.       scanf("%lf", &first);
19.       first_sqrt = sqrt(first);
20.       printf("The square root of the first number is %.2f\n", first_sqrt);
```

Square Root Program (cont'd)

```
21.       /* Get second number and display its square root. */
22.       printf("Enter the second number> ");
23.       scanf("%lf", &second);
24.       second_sqrt = sqrt(second);
25.       printf("The square root of the second number is %.2f\n", second_sqrt);
26.
27.       /* Display the square root of the sum of the two numbers. */
28.       sum_sqrt = sqrt(first + second);
29.       printf("The square root of the sum of the two numbers is %.2f\n",
30.              sum_sqrt);
31.
32.       return (0);
33. }
```

```
Enter the first number> 9.0
The square root of the first number is 3.00
Enter the second number> 16.0
The square root of the second number is 4.00
The square root of the sum of the two numbers is 5.00
```

Library Functions and Code Resue

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>
```

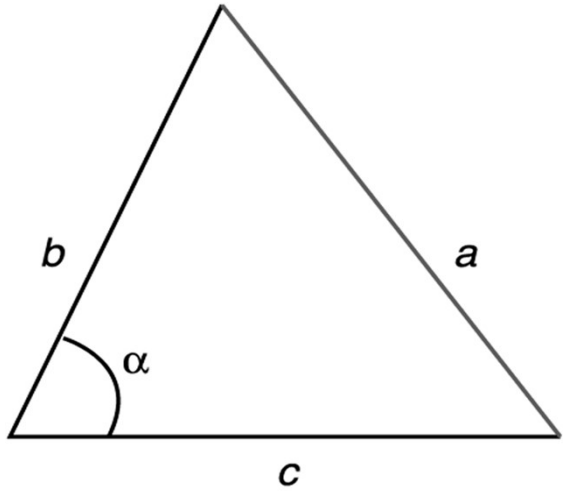
- Computing the roots of: $ax^2 + bx + c = 0$

```
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



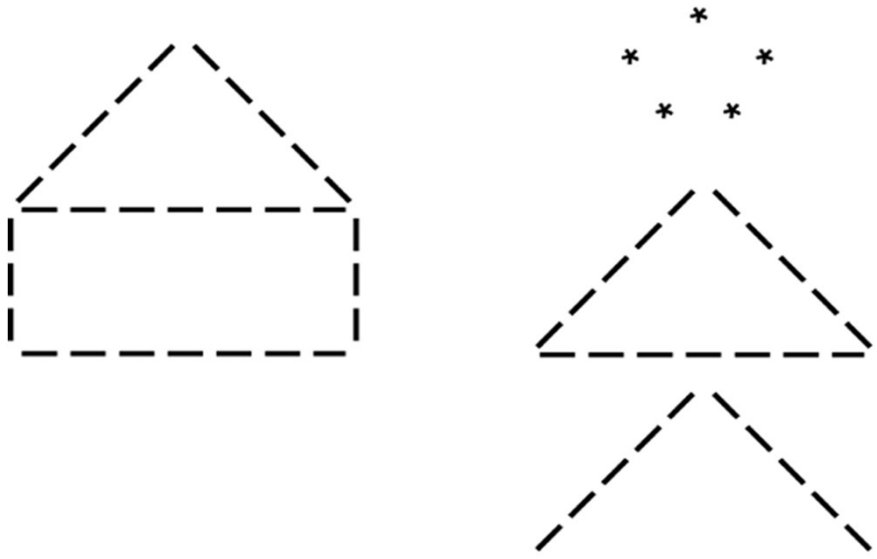
Top-Down Design

- 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

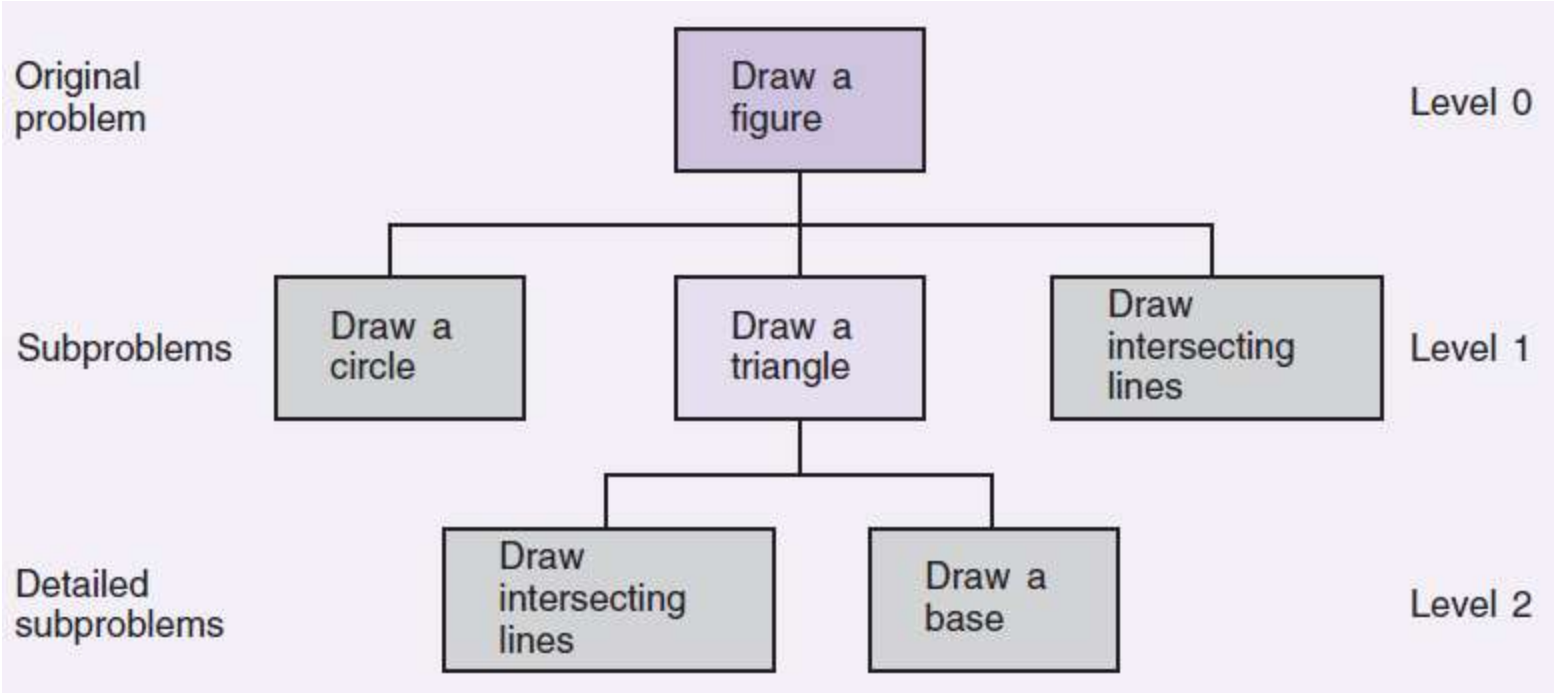
  Simple

  Diagrams



Top-Down Design

- 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, No result */
```

- To draw a triangle, call the function:

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

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

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, No result */
```

- To draw a triangle, call the function:

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

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

Functions 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`**

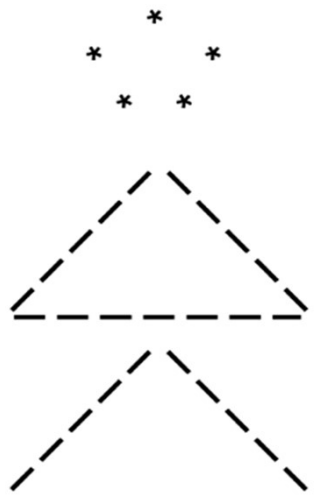
```
void   draw_circle(void);
double sqrt(double x);
```

Functions Prototypes

```
1.  /*
2.   * Draws a stick figure
3.   */
4.
5.  #include <stdio.h>              /* printf definition */
6.
7.  /* function prototypes */
8.
9.  void draw_circle(void);         /* Draws a circle              */
10.
11. void draw_intersect(void);      /* Draws intersecting lines    */
12.
13. void draw_base(void);           /* Draws a base line           */
14.
15. void draw_triangle(void);       /* Draws a triangle            */
16.
17. int
18. main(void)
19. {
20.         /* Draw a circle. */
21.         draw_circle();
22.
23.         /* Draw a triangle. */
24.         draw_triangle();
25.
26.         /* Draw intersecting lines. */
27.         draw_intersect();
28.
29.         return (0);
30. }
```

Before main function

Draws This Stick Figure



Output a String with `printf`

- 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");
}
```

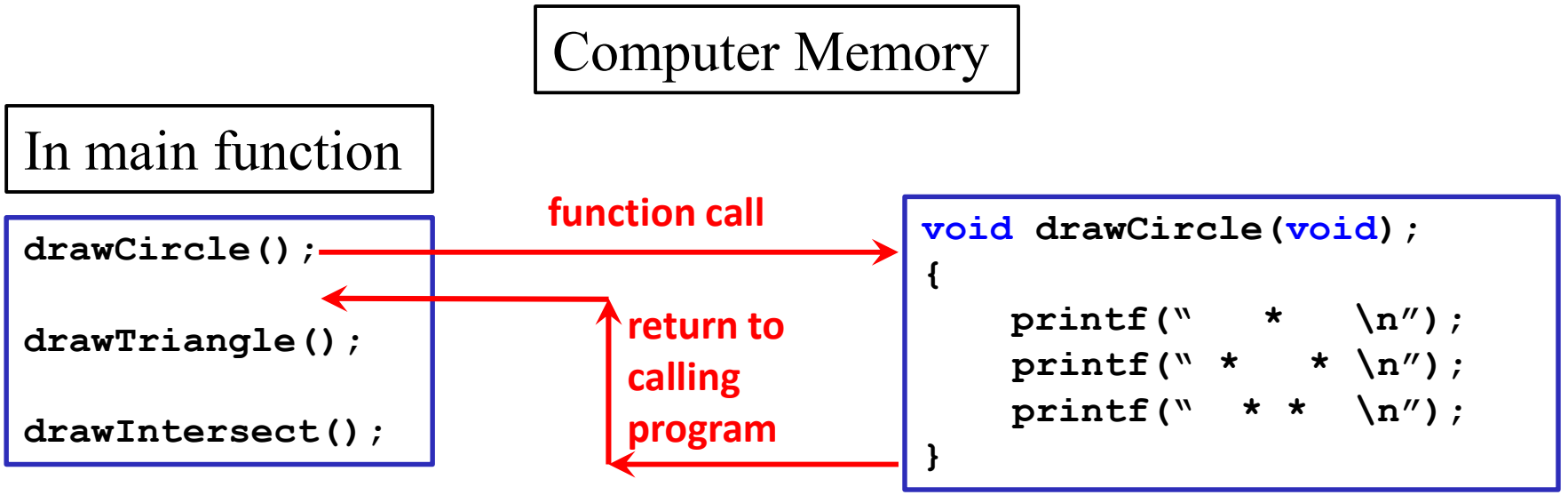
```
/*
 * Draws a triangle
 */
void
draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

Placement of 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



Functions and 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 (void) and no value returned (void)
  - Input arguments, but no value returned (void)
  - 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);
```

Call `print_rboxed` with `rnum = 135.68`

Sample Run

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

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

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

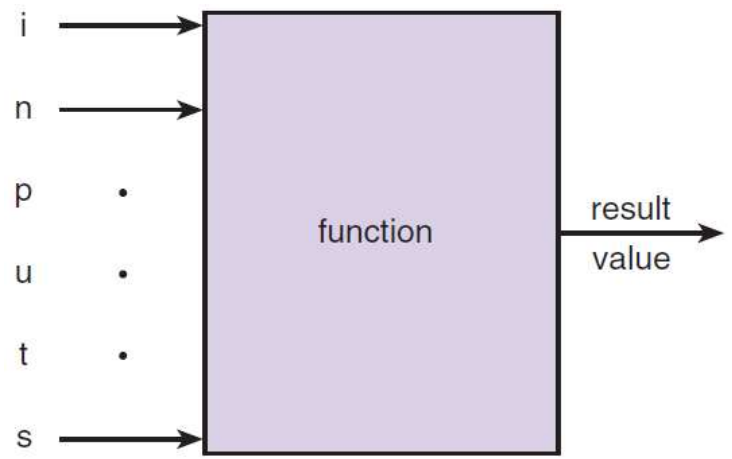
  Example: `print_rbox(x+y); /* function call */`

  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`

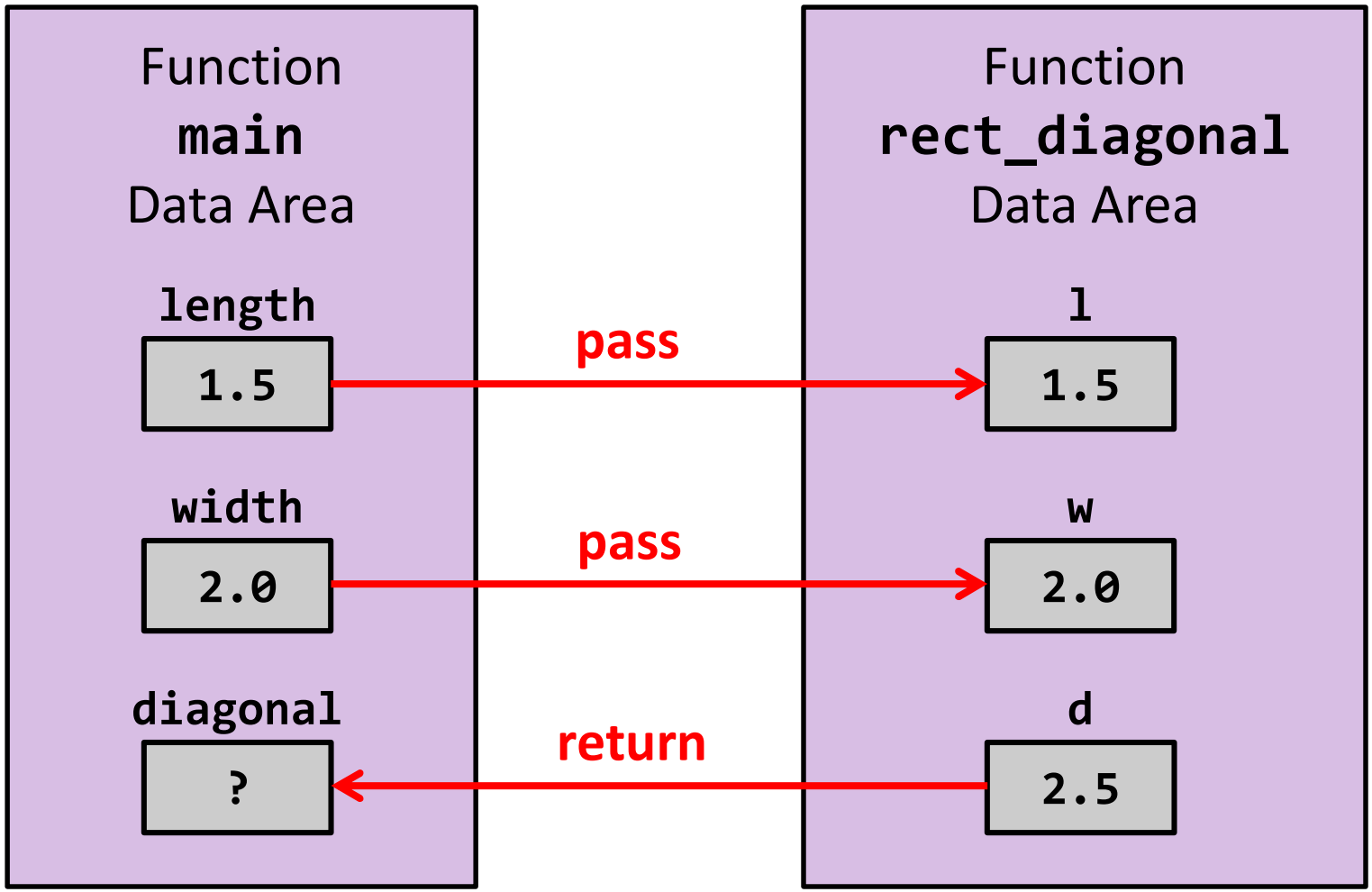
```
/* Testing rect_diagonal function */

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;

}
```

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);
```



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.

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 that you want

Common Programming Error

- 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)`**