Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 6: Functions

Introduction to Computer Programming
(C language)

Nguyễn Tiến Thịnh, Ph.D.

Email: ntthinh@hcmut.edu.vn

2022 – 2023, Semester 1

Course Content

- C.1. Introduction to Computers and Programming
- C.2. C Program Structure and its Components
- C.3. Variables and Basic Data Types
- C.4. Selection Statements
- C.5. Repetition Statements
- **C.6. Functions**
- C.7. Arrays
- C.8. Pointers
- C.9. File Processing

References

- [1] "*C: How to Program*", 7th Ed. – Paul Deitel and Harvey Deitel, Prentice Hall, 2012.
- [2] "*The C Programming Language*", 2nd Ed. – Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988
- and others, especially those on the Internet

Content

- Introduction
- Functions in the standard library
- An example of a function
- Components of a function
- Function call
- Recursion
- Summary

Introduction

- In the previous chapters, we have used several so-called functions in the library:
  - printf in stdio.h
  - scanf in stdio.h
  - fflush in stdio.h
  - sqrt in math.h
  - pow in math.h
  - system in stdlib.h
  - strcmp in string.h
  - strcpy in string.h

Those functions are modular processing units that are:

→Responsible for a certain task

→Reusable in many various programs

Functions in the standard library

- <assert.h>
- <ctype.h>
- <errno.h>
- <float.h>
- <limits.h>
- <locale.h>
- <math.h>
- <setjmp.h>
- <signal.h>
- <stdarg.h>
- <stddef.h>
- <stdio.h>
- <stdlib.h>
- <string.h>
- <time.h>

Source: www.tutorialspoint.com

Functions in the standard library

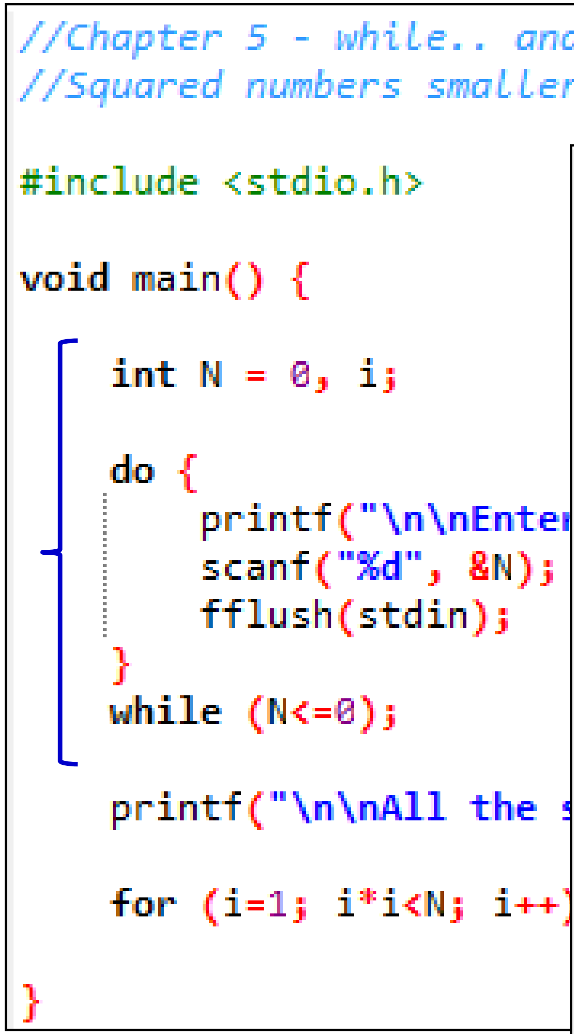
- Some functions in <stdio.h>
  - int printf(const char *format, ...)
    - Sends formatted output to stdout
  - int scanf(const char *format, ...)
    - Reads formatted input from stdin
  - int getchar(void)
    - Gets a character (an unsigned char) from stdin
  - char *gets(char *str)
    - Reads a line from stdin and stores it into the string pointed to, by str. It stops when either the newline character ('\n') is read or when the end-of-file (EOF) is reached, whichever comes first.

Functions in the standard library

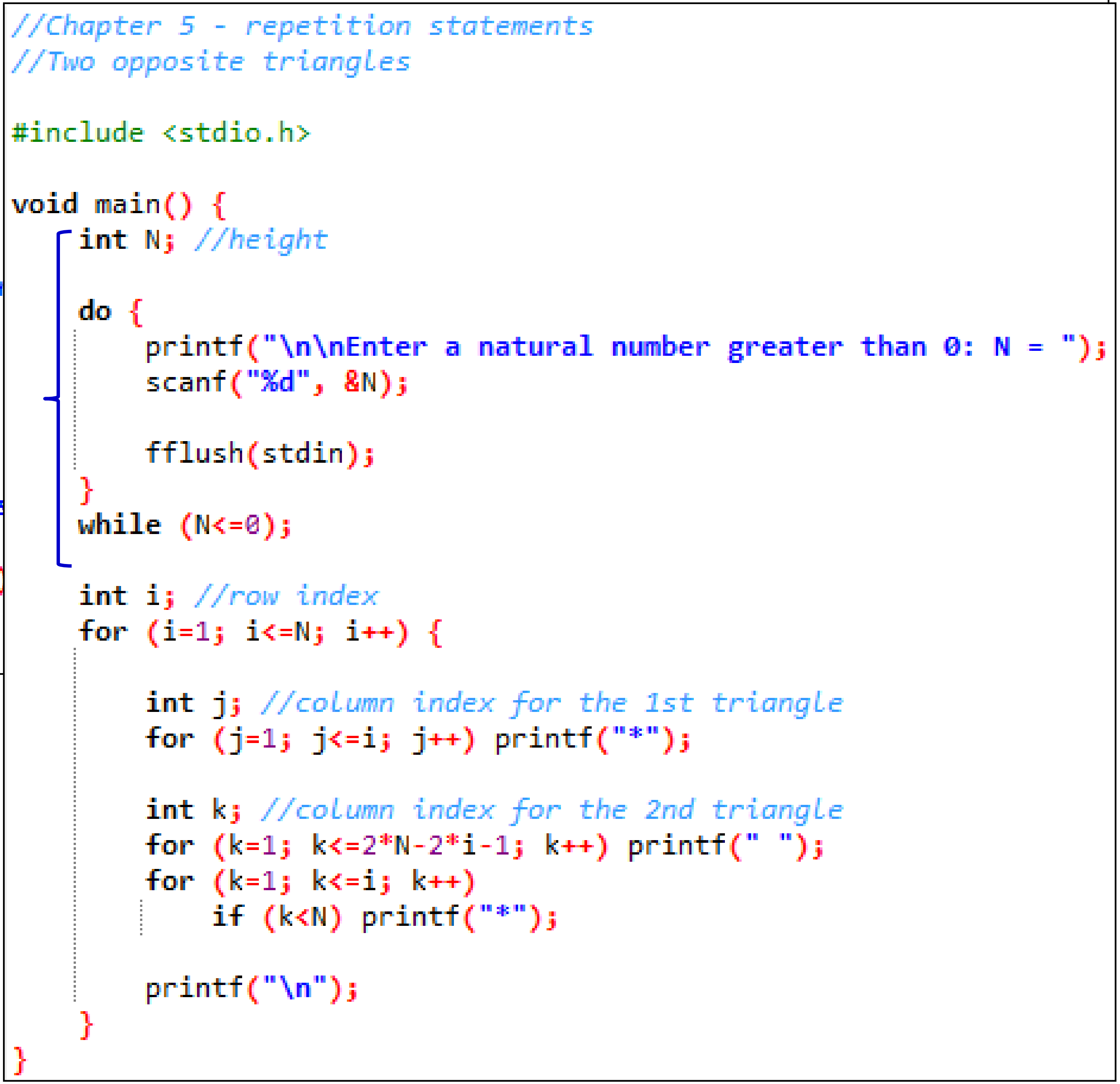
- Some functions in <math.h>
  - double cos(double x)
    - Returns the cosine of a radian angle x
  - double pow(double x, double y)
    - Returns **x** raised to the power of **y**
  - double sqrt(double x)
    - Returns the square root of **x**
  - double ceil(double x)
    - Returns the smallest integer value greater than or equal to **x**
  - double floor(double x)
    - Returns the largest integer value less than or equal to **x**

Functions in the standard library

- Some functions in <stdlib.h>
  - void *malloc(size_t size)
    - Allocates the requested memory and returns a pointer to it
  - void free(void *ptr)
    - Deallocates the memory previously allocated by a call to *calloc, malloc,* or *realloc*
  - int rand(void)
    - Returns a pseudo-random number in the range of 0 to *RAND_MAX* (at least 32767, up to implementation)
  - int system(const char *string)
    - The command specified by string is passed to the host environment to be executed by the command processor
      - E.g. “pause”, “cls”, “date”



```
//Chapter 5 - while.. and for.. statements
//Squared numbers smaller than N which is input by a user

#include <stdio.h>

void main() {

    int N = 0, i;

    do {
        printf("\n\nEnter
        scanf("%d", &N);
        fflush(stdin);
    }
    while (N<=0);

    printf("\n\nAll the

    for (i=1; i*i<N; i++)

}
```



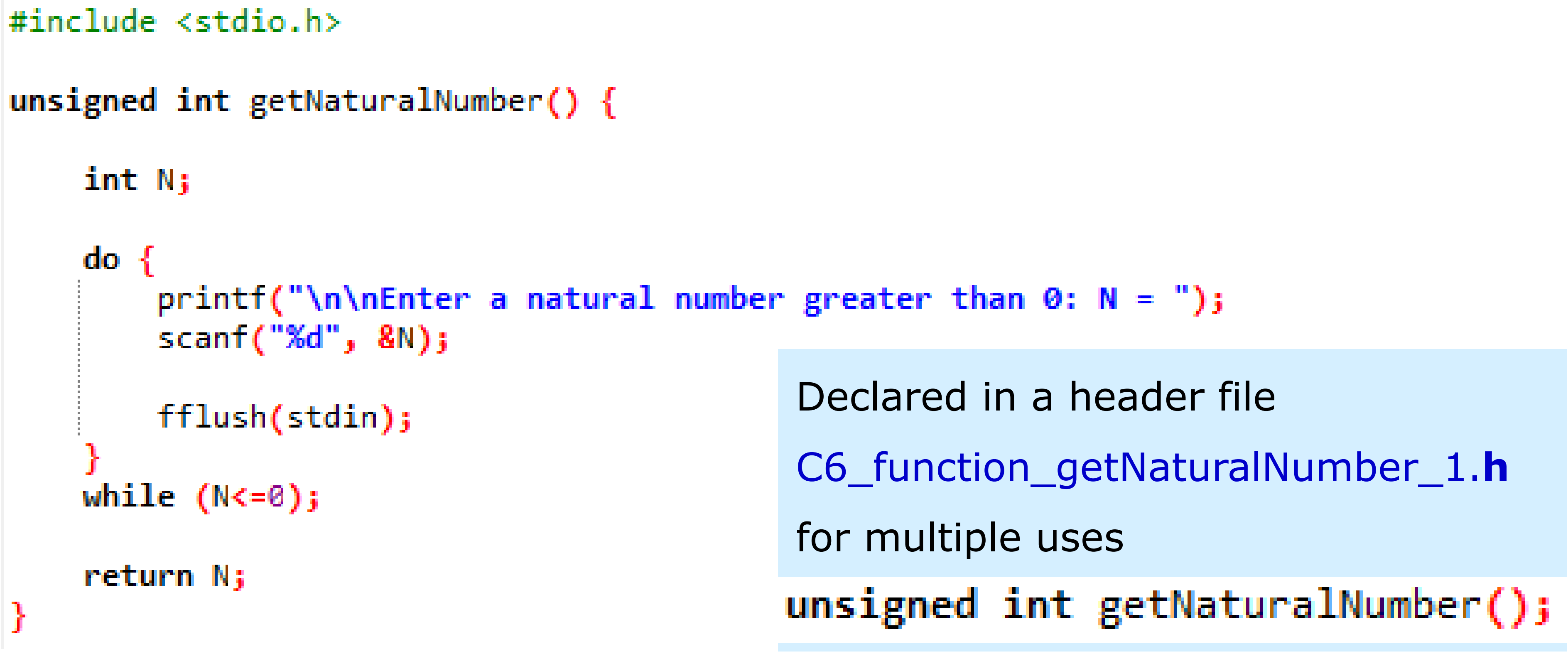
```
//Chapter 5 - repetition statements
//Two opposite triangles

#include <stdio.h>

void main() {
    int N; //height

    do {
        printf("\n\nEnter a natural number greater than 0: N = ");
        scanf("%d", &N);

        fflush(stdin);
    }
    while (N<=0);

    int i; //row index
    for (i=1; i<=N; i++) {

        int j; //column index for the 1st triangle
        for (j=1; j<=i; j++) printf("*");

        int k; //column index for the 2nd triangle
        for (k=1; k<=2*N-2*i-1; k++) printf(" ");
        for (k=1; k<=i; k++)
            if (k<N) printf("*");

        printf("\n");
    }
}
```

Repeated code!!!

Can we just code them once and then make use of them over the time just like those in the standard library?

Introductions

- Let’s define a function: getNaturalNumber()



```
#include <stdio.h>

unsigned int getNaturalNumber() {

    int N;

    do {
        printf("\n\nEnter a natural number greater than 0: N = ");
        scanf("%d", &N);

        fflush(stdin);
    }
    while (N<=0);

    return N;
}
```

Declared in a header file

C6_function_getNaturalNumber_1.**h**

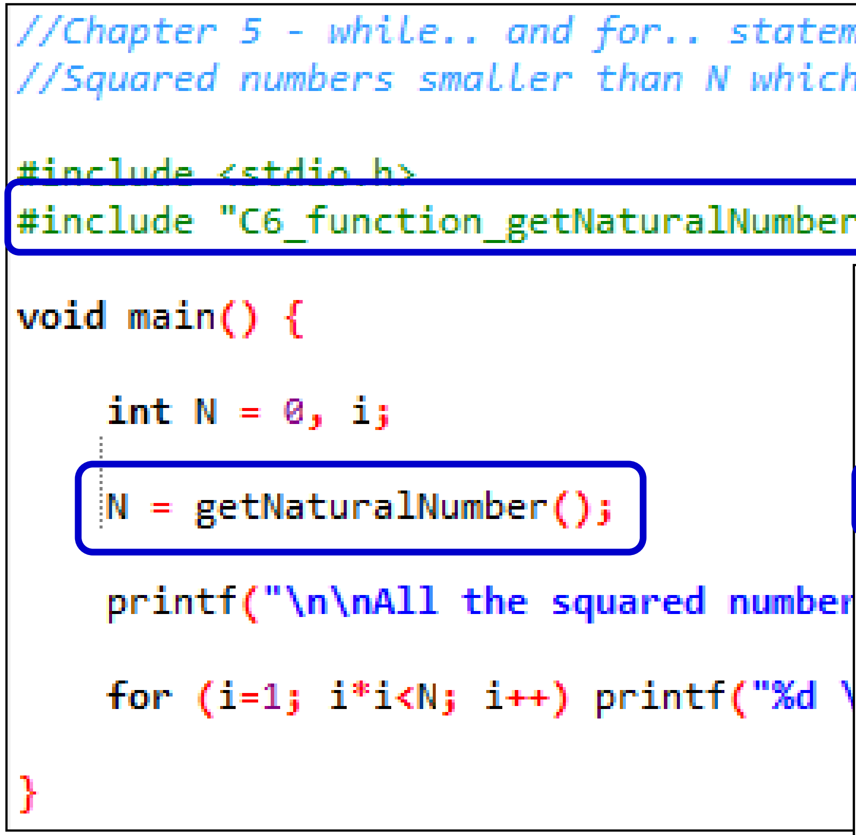
for multiple uses

```
unsigned int getNaturalNumber();
```

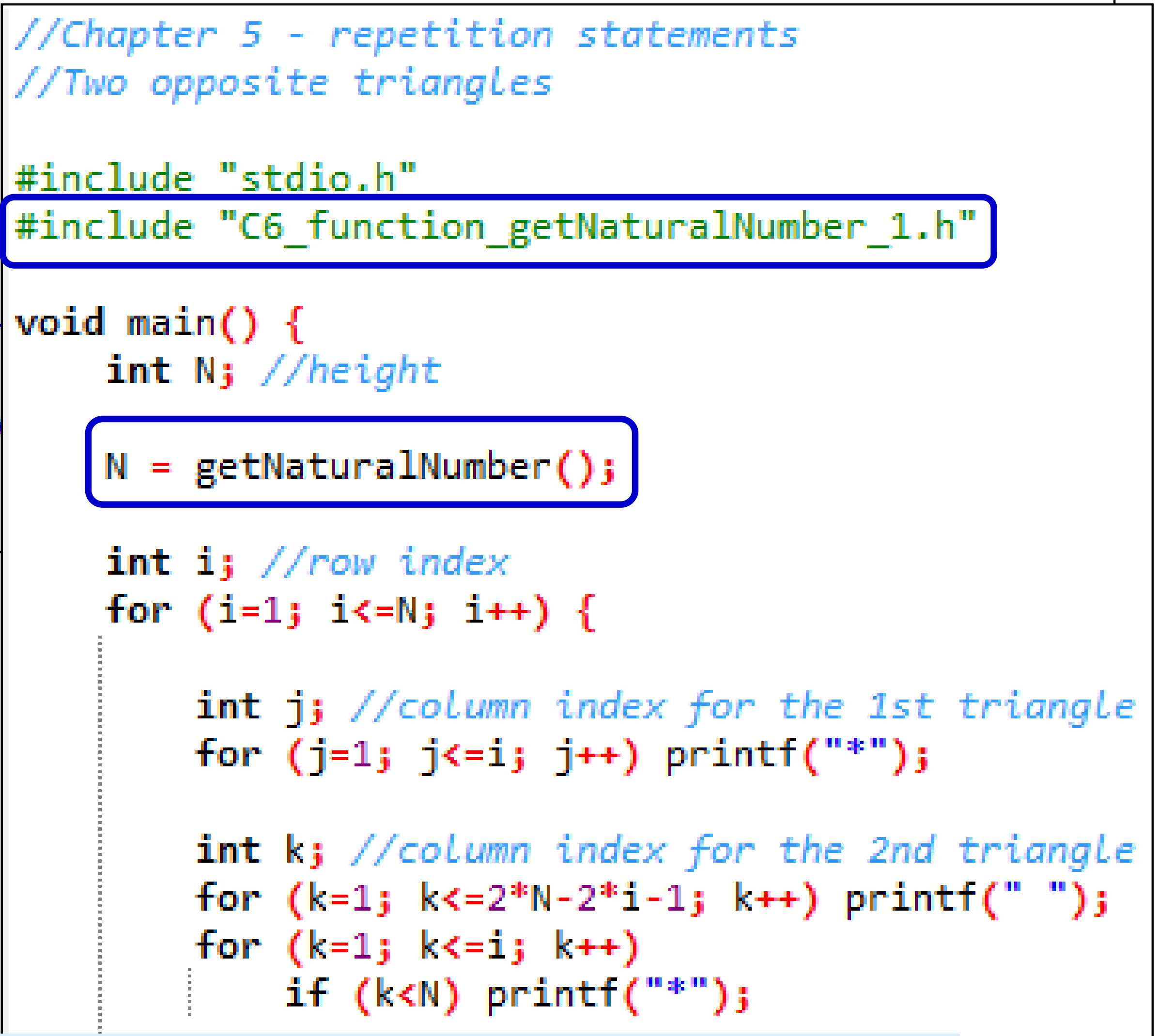
Source code file: C6_function_getNaturalNumber.c

Purpose: to ask users to input a natural number

until a valid number is input



```
//Chapter 5 - while.. and for.. statements
//Squared numbers smaller than N which is input by a user

#include <stdio.h>
#include "C6_function_getNaturalNumber_1.h"

void main() {

    int N = 0, i;

    N = getNaturalNumber();

    printf("\n\nAll the squared number

    for (i=1; i*i<N; i++) printf("%d \

}
```



```
//Chapter 5 - repetition statements
//Two opposite triangles

#include "stdio.h"
#include "C6_function_getNaturalNumber_1.h"

void main() {
    int N; //height

    N = getNaturalNumber();

    int i; //row index
    for (i=1; i<=N; i++) {

        int j; //column index for the 1st triangle
        for (j=1; j<=i; j++) printf("*");

        int k; //column index for the 2nd triangle
        for (k=1; k<=2*N-2*i-1; k++) printf(" ");
        for (k=1; k<=i; k++)
            if (k<N) printf("*");
```

Use of our previously defined function, which is declared in a header file:

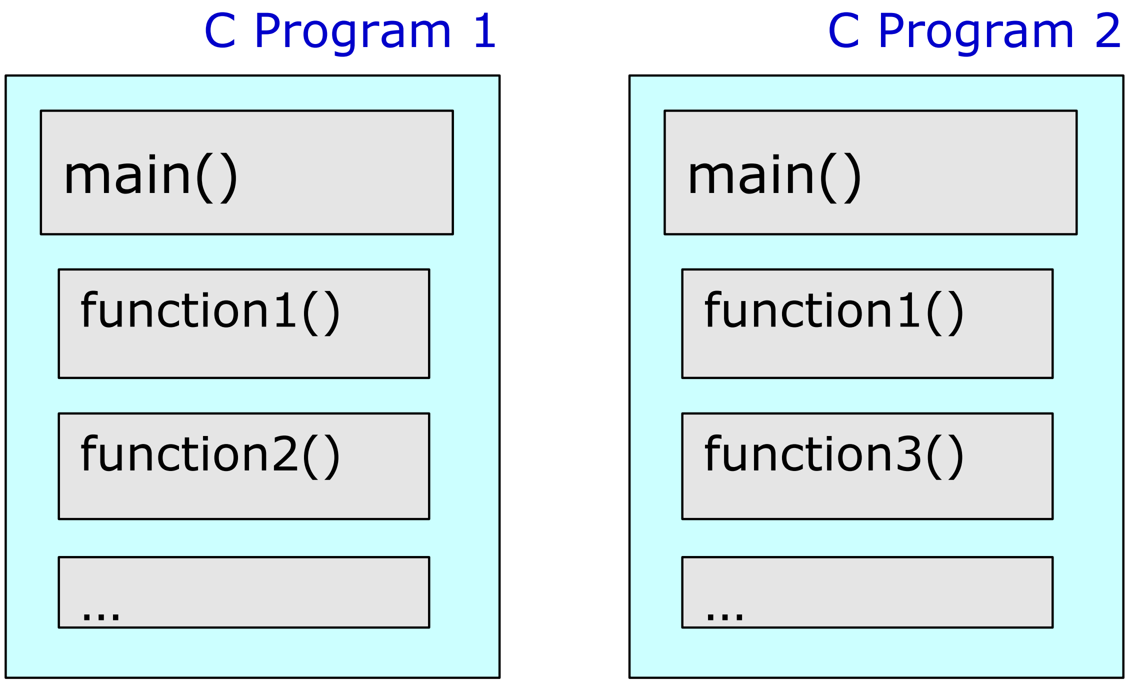
"C6_function_getNaturalNumber_1.h"

Compile:

gcc C6_starTriangle_function_1.c C6_function_getNaturalNumber.c –o C6_starTriangle_function_1.exe

Introduction

- A function
  - A processing unit to perform a specific task
  - A means to modularize a program for manageable program development



C Program 1

main()

function1()

function2()

…

C Program 2

main()

function1()

function3()

…

Divide-and-conquer

Reusable

Information hiding

Abstraction

Easy for debugging

Introduction

- Issues related to functions
  - Function definition
  - Function declaration
  - Function call

An example of a function

- Prepare your own library for numbers
  - Compute the sum of N first natural numbers
    - sum = 1 + 2 + 3 + … + (N-1) + N
  - Compute the factorial of N, a natural number
    - factorial = 1*2*3*…*(N-1)*N
  - Compute the n-th power of x, a floating-point number
    - x^n = x*x*x*…*x
  - Count the number of digits in N, a natural number
    - N = 123456 => Number of digits = 6
  - Round x, a floating-point number, with two digits after the decimal point
    - x = 1.23456 => x = 1.23
    - x = 9.87654321 => x = 9.88

An example of a function

- Prepare your own library for numbers
  - Check if a natural number is a prime number
    - 7 => true (≠0)
    - 8 => false (0)
  - Check if a natural number is a squared number
    - 4 => true (≠0)
    - 8 => false (0)
  - Toggle non-zero digits to ‘9’ digits in an integer number to generate its 9-based mask
    - 113789 => 999999
    - -10789 => -90999
  - Count the number of occurrences of each digit in an integer number
    - 113789 => 0: 0; 1: 2; 2: 0; 3: 1; 4: 0; 5: 0; 6: 0; 7: 1; 8: 1; 9: 1
    - -20054 => 0: 2; 1: 1; 2: 1; 3: 0; 4: 1; 5: 1; 6: 0; 7: 0; 8: 0; 9: 0

An example of a function

- Prepare your own library for numbers
  - Estimate a value of e, the natural number

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.71828$$

  - Estimate a value of e^x

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

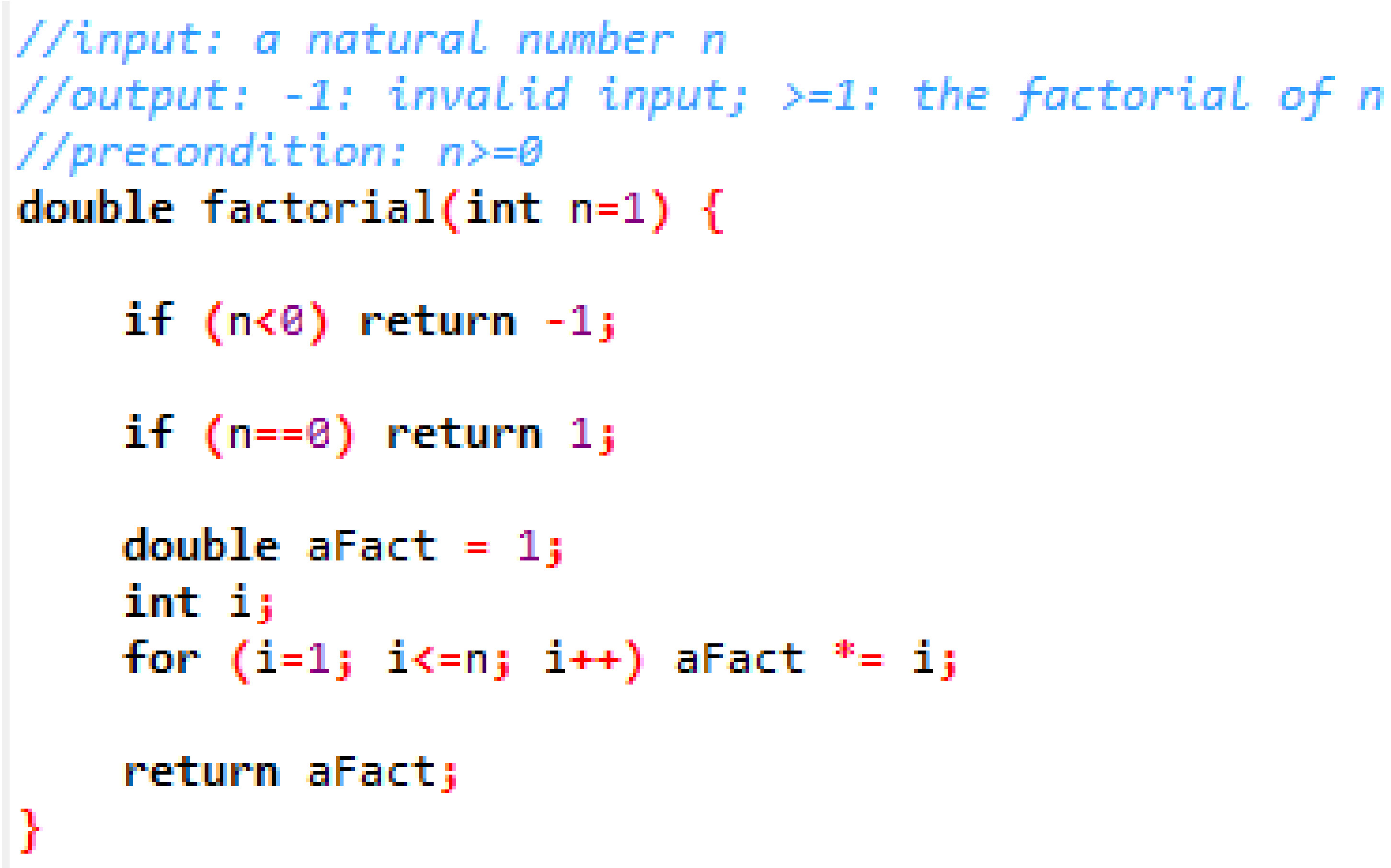
  - Estimate a value of PI

$$\pi = \sum_{k=0}^{\infty} \frac{(-1)^k 4}{2k+1}$$

  - …

Components of a function

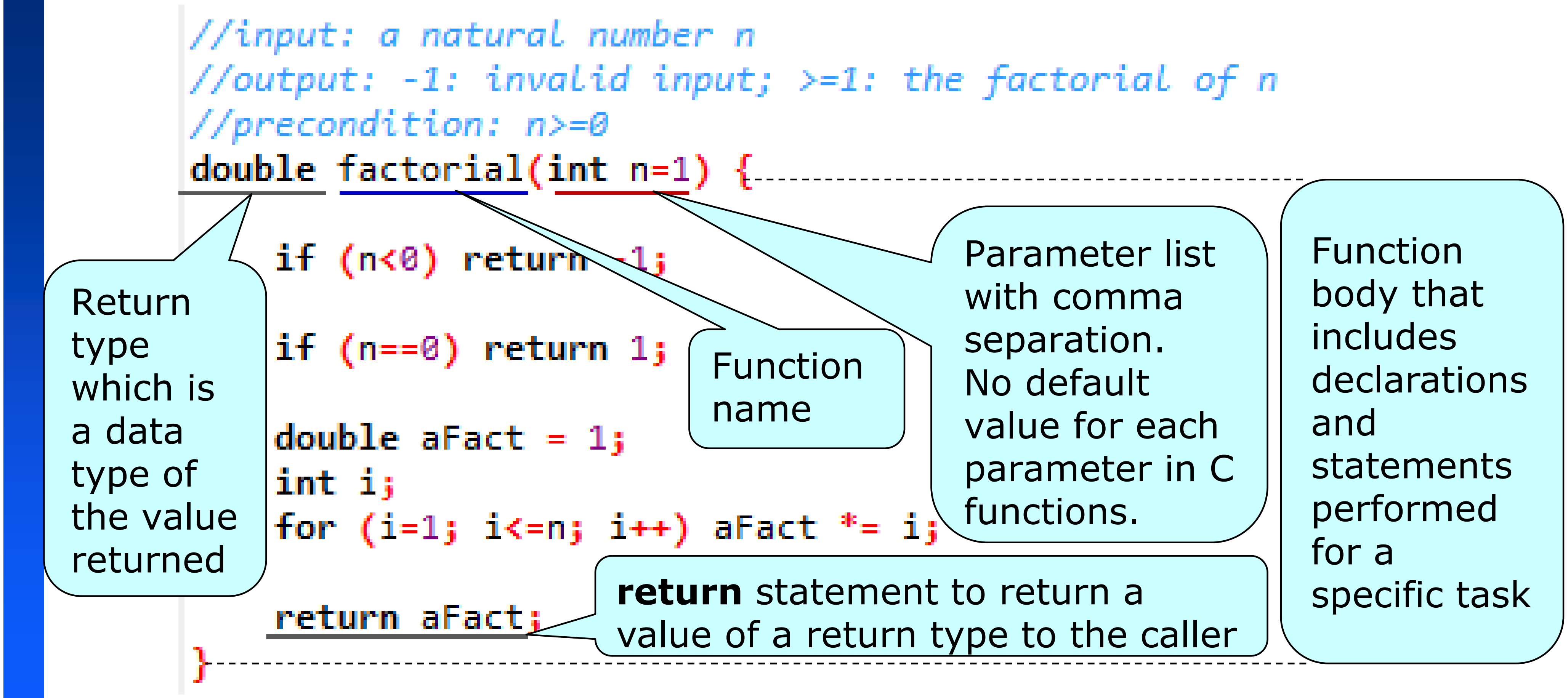
- Given N, a natural number, calculate the *factorial of N*: N! = 1*2*..*N = (N-1)!*N



```
//input: a natural number n
//output: -1: invalid input; >=1: the factorial of n
//precondition: n>=0
double factorial(int n=1) {

    if (n<0) return -1;

    if (n==0) return 1;

    double aFact = 1;
    int i;
    for (i=1; i<=n; i++) aFact *= i;

    return aFact;
}
```

Components of a function

- Given N, a natural number, calculate the *factorial of N*: N! = 1*2*..*N = (N-1)!*N



```
//input: a natural number n
//output: -1: invalid input; >=1: the factorial of n
//precondition: n>=0
double factorial(int n=1) {

    if (n<0) return -1;

    if (n==0) return 1;

    double aFact = 1;
    int i;
    for (i=1; i<=n; i++) aFact *= i;

    return aFact;
}
```

Return type which is a data type of the value returned

Function name

Parameter list with comma separation. No default value for each parameter in C functions.

Function body that includes declarations and statements performed for a specific task

return statement to return a value of a return type to the caller

Components of a function

[*static*] *return-type function-name* **(***argument-declarations***)**
{
        declarations and statements
}

- *function-name*: a valid identifier

Part of the input

- *argument-declarations*: a list of formal parameters in communication
  + Each parameter is regarded as a local variable with a data type.
  + Each parameter can be specified with "**const**" for unchanged intention.
  + Each parameter can be passed by value or by reference if it is a pointer.

Processing in its body

- *declarations*: a list of local variables
  + Each variable can be **auto** or **static** with one single initialization.
- *statements*: zero, one, or many statements of any kind

Part of the output

- *return-type*: a valid data type or **void**
  + Statement **return** [<expression>]; in the body is used to end the called function and return [a value] to its caller. If not, close brace of the body ends the called function and program control is switched to its caller.

Characteristic

- [***static***]: optional specification to make the function available only in the file where it is defined.

Concepts related to functions

- *Function definition*:

  [*static*] *return-type function-name* **(***argument-declarations***)**
  {
  declarations and statements
  }

- *Function prototype*:

  return-type function-name **(***argument-declarations***);**

- *Function signature*:

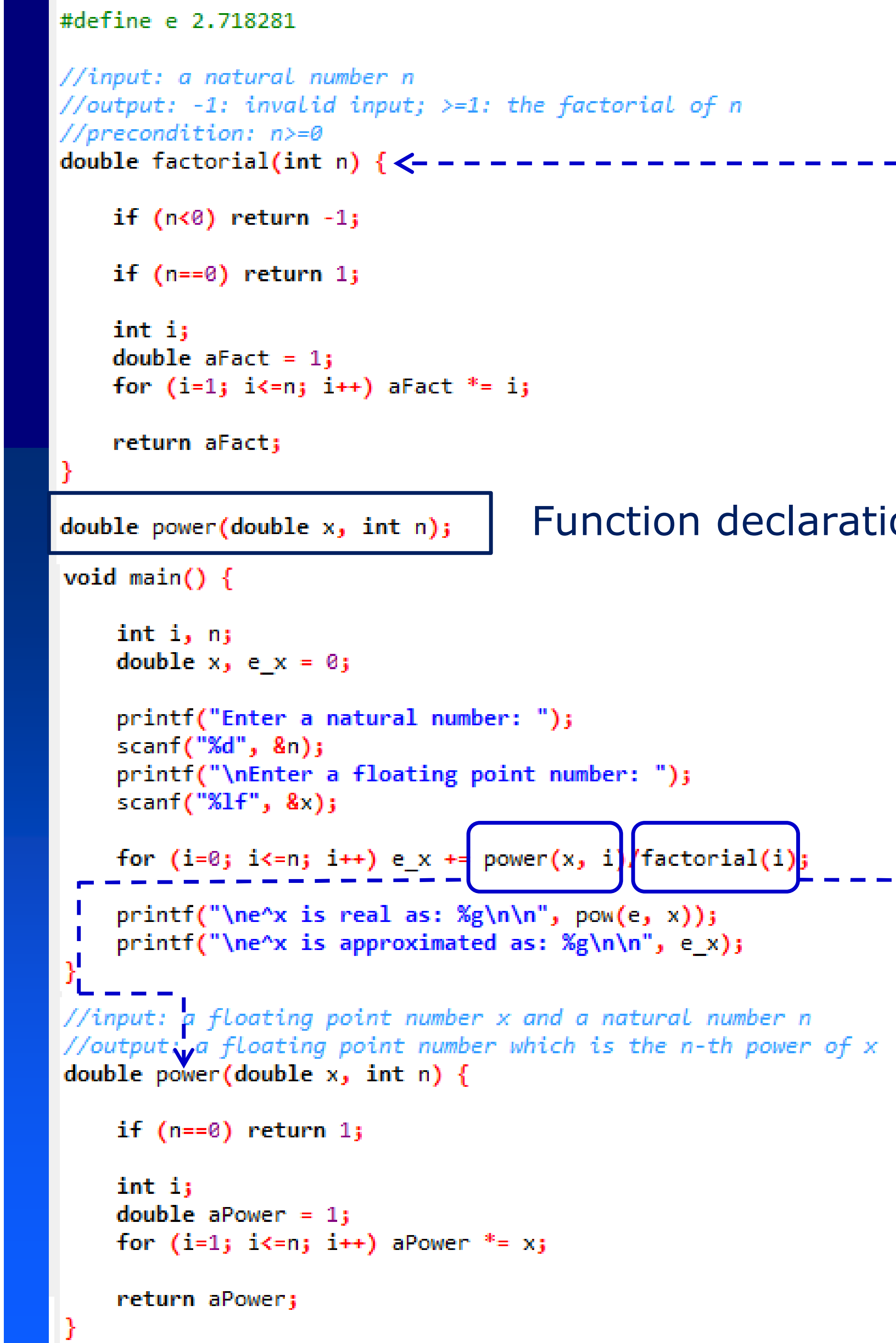
  function-name **(***argument-declarations***)**

- No concept of "nested functions"!
  - Implementation-dependent

Where is a function defined and declared?

- A function definition can be placed in:
  - the same file where the main() function is
    - Before the main() function
    - After the main() function
  - a separated file where the main() function is not
- Regardless of where a function is defined, its declaration is required before any call to it.
  - Function prototype in the global declaration section
  - Function prototype in a header file for common use

A program for an approximation of the **x**-th power of **e**



```
#define e 2.718281

//input: a natural number n
//output: -1: invalid input; >=1: the factorial of n
//precondition: n>=0
double factorial(int n) {

    if (n<0) return -1;

    if (n==0) return 1;

    int i;
    double aFact = 1;
    for (i=1; i<=n; i++) aFact *= i;

    return aFact;
}
double power(double x, int n);
void main() {

    int i, n;
    double x, e_x = 0;

    printf("Enter a natural number: ");
    scanf("%d", &n);
    printf("\nEnter a floating point number: ");
    scanf("%lf", &x);

    for (i=0; i<=n; i++) e_x += power(x, i)/factorial(i);

    printf("\ne^x is real as: %g\n\n", pow(e, x));
    printf("\ne^x is approximated as: %g\n\n", e_x);
}
//input: a floating point number x and a natural number n
//output: a floating point number which is the n-th power of x
double power(double x, int n) {

    if (n==0) return 1;

    int i;
    double aPower = 1;
    for (i=1; i<=n; i++) aPower *= x;

    return aPower;
}
```

Function definition

= Function declaration

(as it is placed before its call)

Function declaration

a call to a function

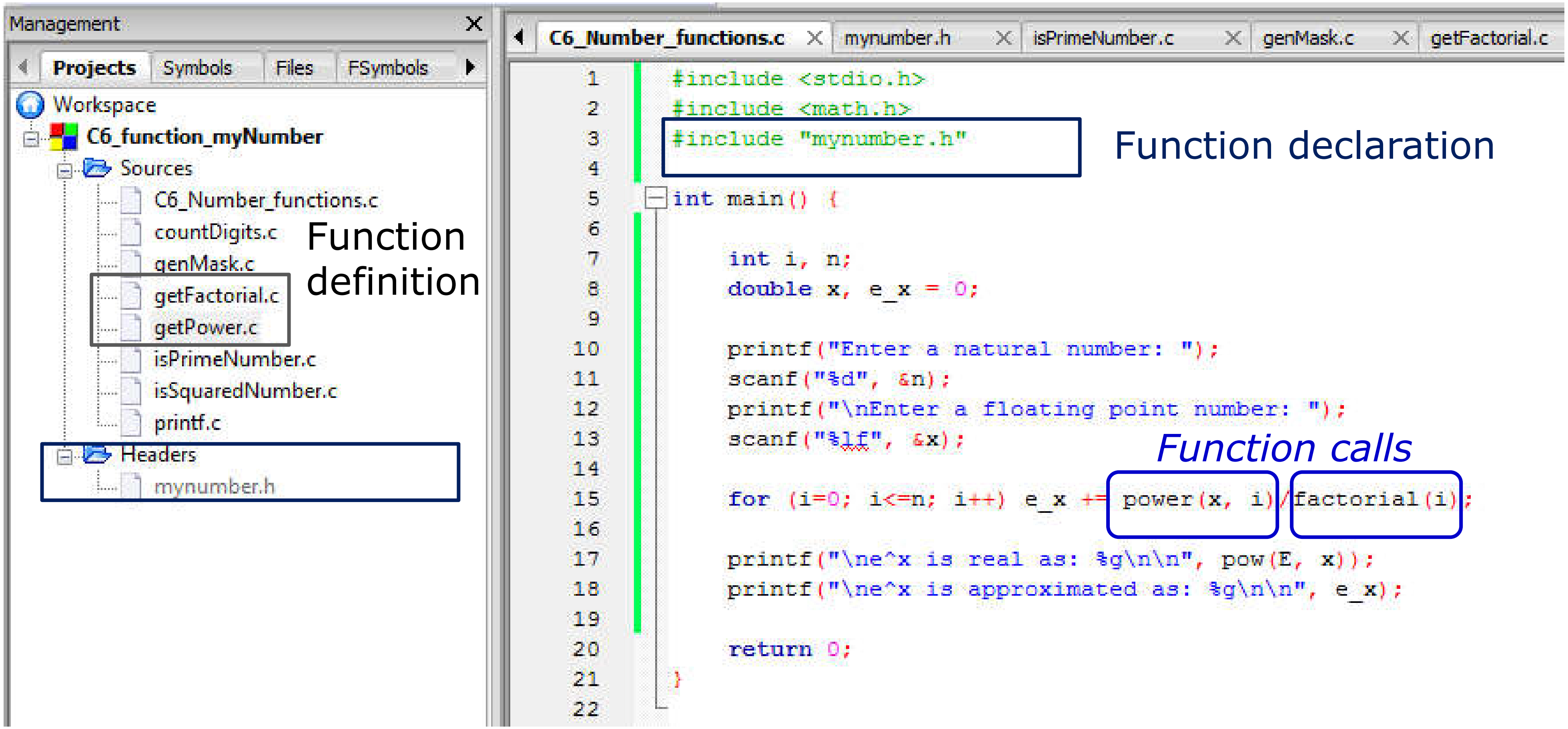
Function definition

(a function declaration is required

as it is placed after its call.)

Where is a function defined and declared?

SEPARATED FILES



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

int main() {

    int i, n;
    double x, e_x = 0;

    printf("Enter a natural number: ");
    scanf("%d", &n);
    printf("\nEnter a floating point number: ");
    scanf("%lf", &x);

    for (i=0; i<=n; i++) e_x += power(x, i)/factorial(i);

    printf("\ne^x is real as: %g\n\n", pow(E, x));
    printf("\ne^x is approximated as: %g\n\n", e_x);

    return 0;
}
```

Function declaration

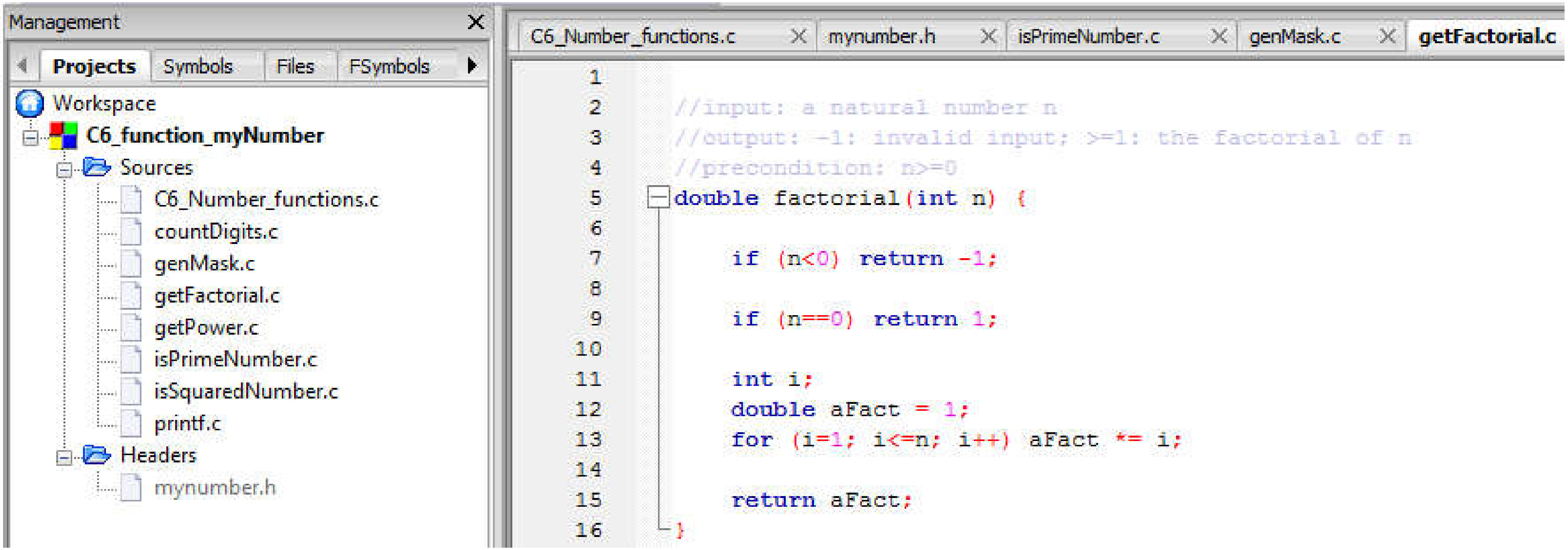
Function definition

Function calls

Where is a function defined and declared?

Source file

getFactorial.c

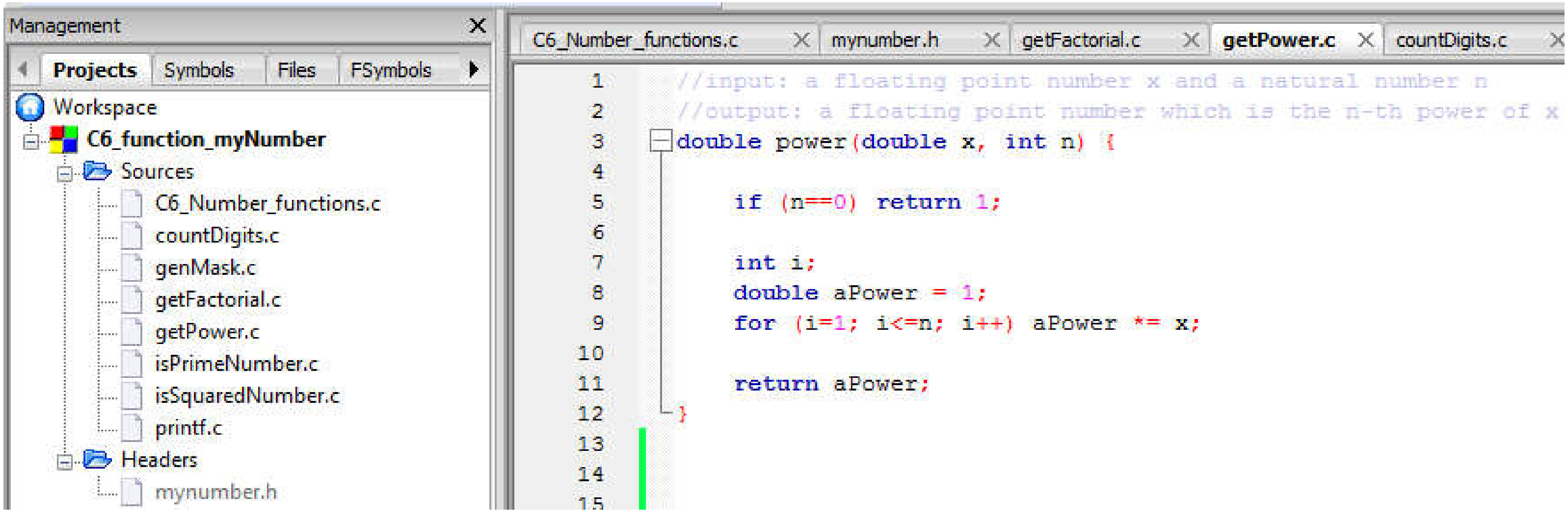


```c

//input: a natural number n
//output: -1: invalid input; >=1: the factorial of n
//precondition: n>=0
double factorial(int n) {

    if (n<0) return -1;

    if (n==0) return 1;

    int i;
    double aFact = 1;
    for (i=1; i<=n; i++) aFact *= i;

    return aFact;
}
```

Where is a function defined and declared?

Source file

getPower.c

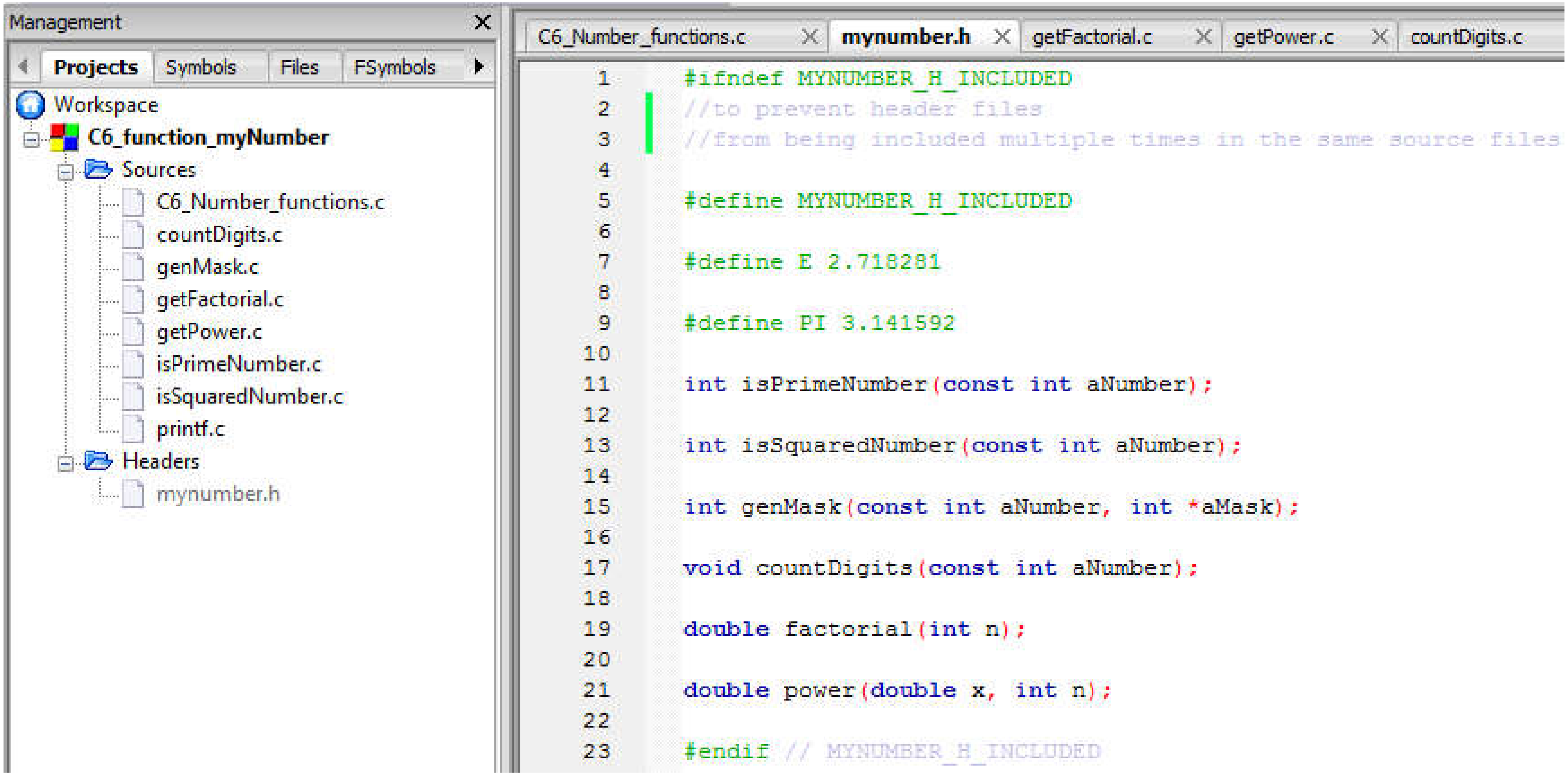


```c
//input: a floating point number x and a natural number n
//output: a floating point number which is the n-th power of x
double power(double x, int n) {

    if (n==0) return 1;

    int i;
    double aPower = 1;
    for (i=1; i<=n; i++) aPower *= x;

    return aPower;
}
```

Where is a function defined and declared?

Header file for common use

mynumber.h



```c
#ifndef MYNUMBER_H_INCLUDED
//to prevent header files
//from being included multiple times in the same source files

#define MYNUMBER_H_INCLUDED

#define E 2.718281

#define PI 3.141592

int isPrimeNumber(const int aNumber);

int isSquaredNumber(const int aNumber);

int genMask(const int aNumber, int *aMask);

void countDigits(const int aNumber);

double factorial(int n);

double power(double x, int n);

#endif // MYNUMBER_H_INCLUDED
```

Function call

- Function call is a mention of a function to another function or itself.
  - The function whose function body contains a a mention is called a calling function or a caller.
  - The function whose name is mentioned in the caller’s function body is called a called function or a callee.
  - The caller is the same as or different from the callee.
- The program in C6_function_myNumber
  - The main() function calls the printf(), scanf(), pow(), factorial(), and power() functions.
    - Caller = main
    - Callees = printf, scanf, pow, factorial, power

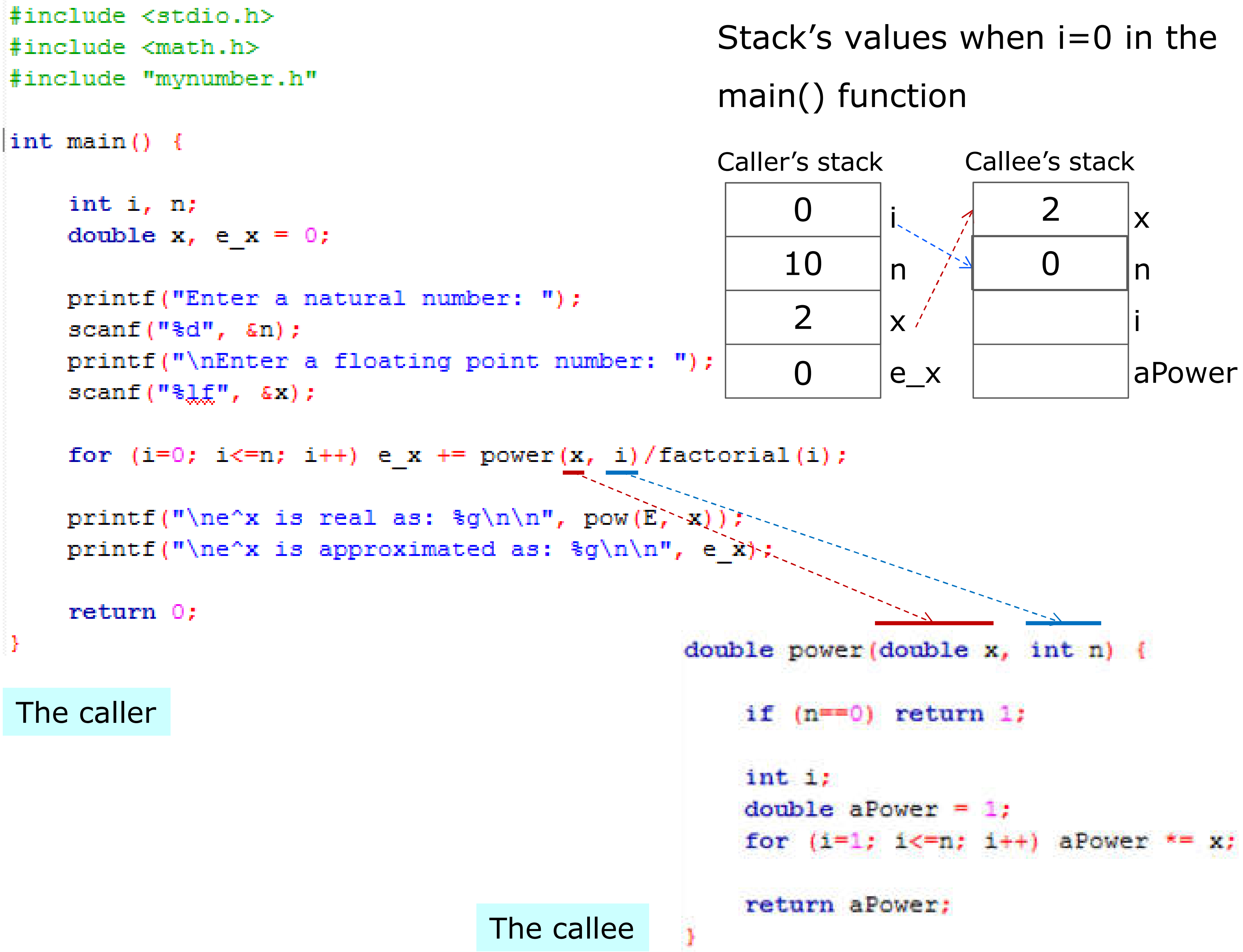
Function call

- A function call is mentioned in the function body of the caller as:

  function-name (*argument-list*)

  - *argument-list*
    - Optional, i.e. empty if the callee has no argument
    - Each argument is called actual parameter which is an expression corresponding to a formal parameter by order.
    - Each argument has a type compatible with the type of its corresponding formal parameter. Otherwise, type conversion with promotion or truncation is performed.
    - Assignment of each expression value to the corresponding formal parameter's memory is performed.
  - A value of a return type (if it is not **void**) is returned via this function call.

Stack's values when i=0 in the main() function



```c
#include <stdio.h>
#include <math.h>
#include "mynumber.h"

int main() {

    int i, n;
    double x, e_x = 0;

    printf("Enter a natural number: ");
    scanf("%d", &n);
    printf("\nEnter a floating point number: ");
    scanf("%lf", &x);

    for (i=0; i<=n; i++) e_x += power(x, i)/factorial(i);

    printf("\ne^x is real as: %g\n\n", pow(E, x));
    printf("\ne^x is approximated as: %g\n\n", e_x);

    return 0;
}
```

The caller

Caller's stack

0	i
10	n
2	x
0	e_x

Callee's stack

2	x
0	n
	i
	aPower

```c
double power(double x, int n) {

    if (n==0) return 1;

    int i;
    double aPower = 1;
    for (i=1; i<=n; i++) aPower *= x;

    return aPower;
}
```

The callee

Function call

- Function call by value
  - Parameters are passed by value.
    - The actual parameter values are copied into local storage of the formal parameters of the callee.
  - The caller and callee do not share any memory.
- Function call by reference
  - C has no explicit reference parameters but implements them via pointers, i.e. address passing.
  - Pointers are passed to the arguments.
    - There is only one copy of the value at any time.
  - The caller and callee have access to the value in their shared memory through pointers.

Function call

A function to swap two integer numbers

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

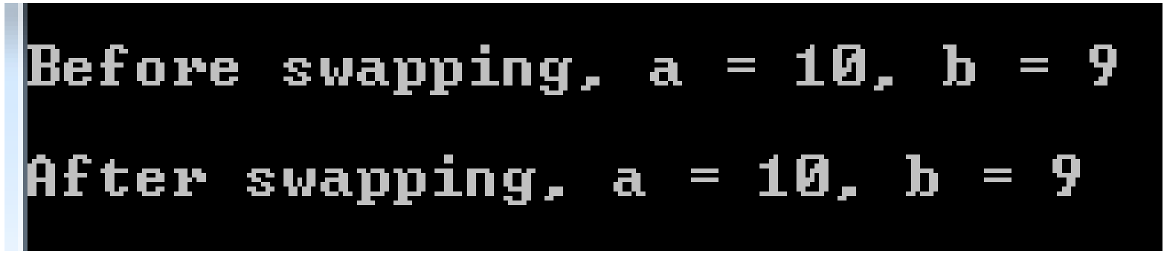
a and **b** will be passed by values of int type.

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

a and **b** will be passed by pointers to int values, i.e. addresses of the memory that contains int values.

Function call by value

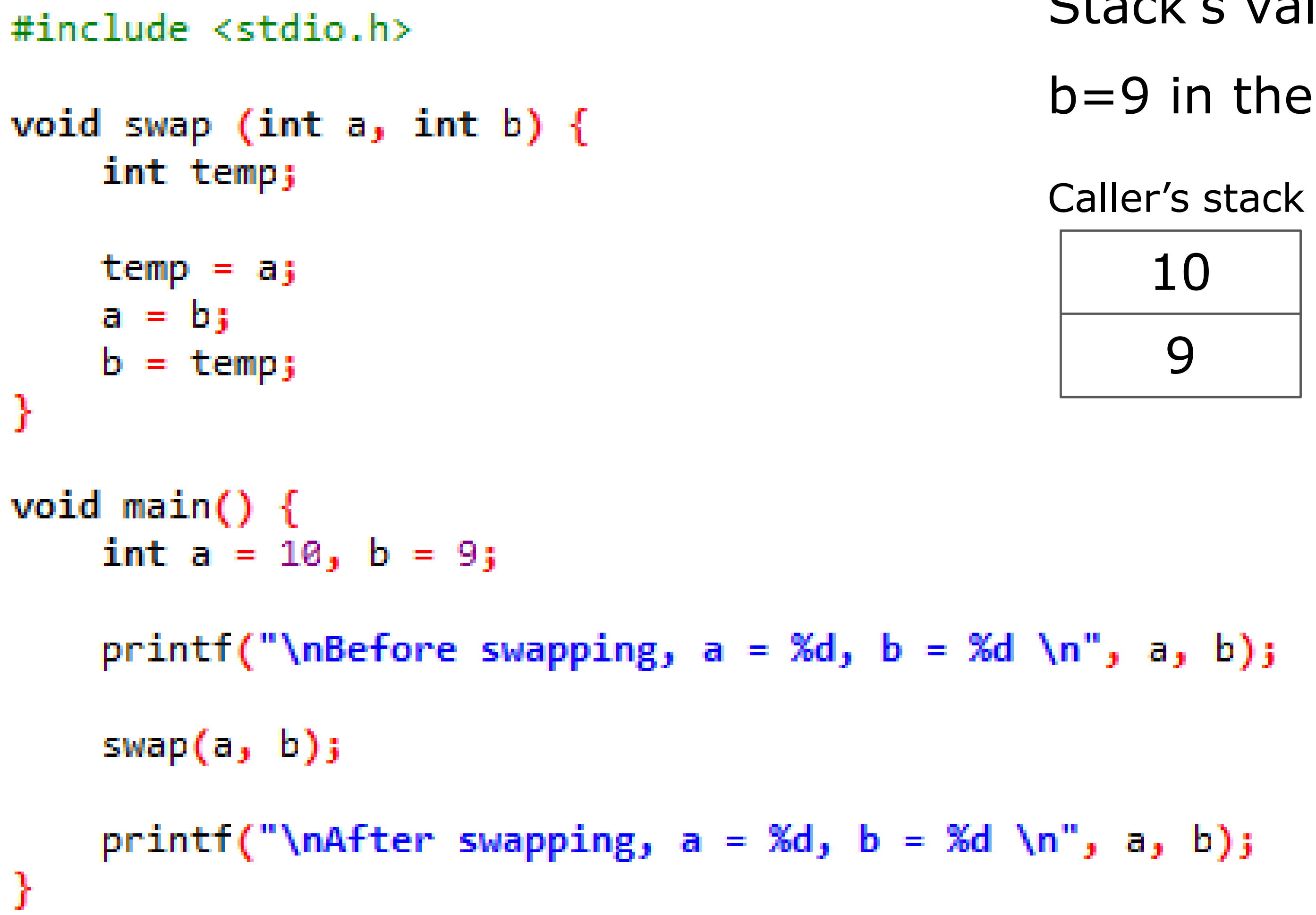
```
#include <stdio.h>

void swap (int a, int b) {
    int temp;

    temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 10, b = 9;

    printf("\nBefore swapping, a = %d, b = %d \n", a, b);

    swap(a, b);

    printf("\nAfter swapping, a = %d, b = %d \n", a, b);
}
```



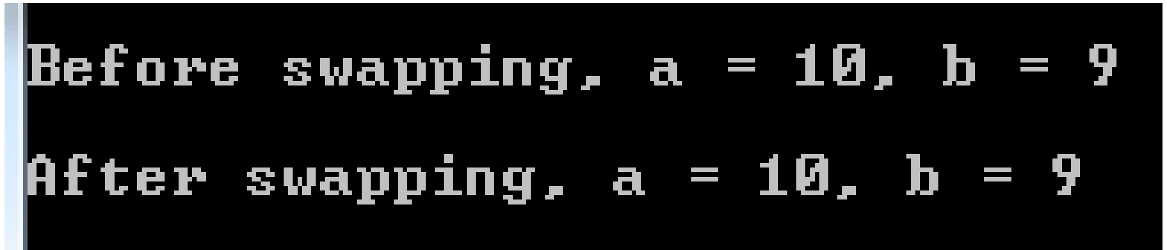
```
Before swapping, a = 10, b = 9

After swapping, a = 10, b = 9
```

- Change on formal parameters in the callee has no impact on actual parameters in the caller.

Function call by value



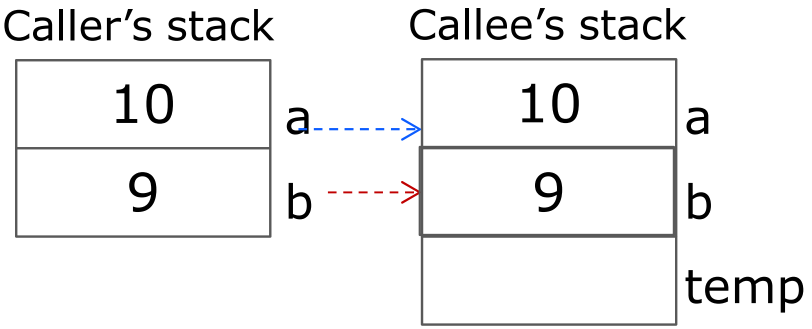
```c
#include <stdio.h>

void swap (int a, int b) {
    int temp;

    temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 10, b = 9;

    printf("\nBefore swapping, a = %d, b = %d \n", a, b);

    swap(a, b);

    printf("\nAfter swapping, a = %d, b = %d \n", a, b);
}
```



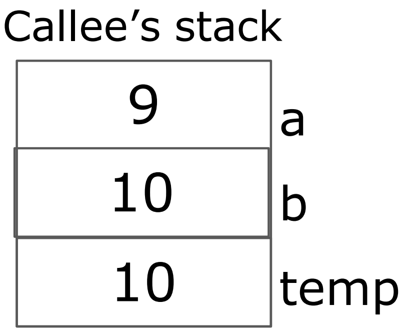
```
Before swapping, a = 10, b = 9

After swapping, a = 10, b = 9
```

Stack's values when a=10 and

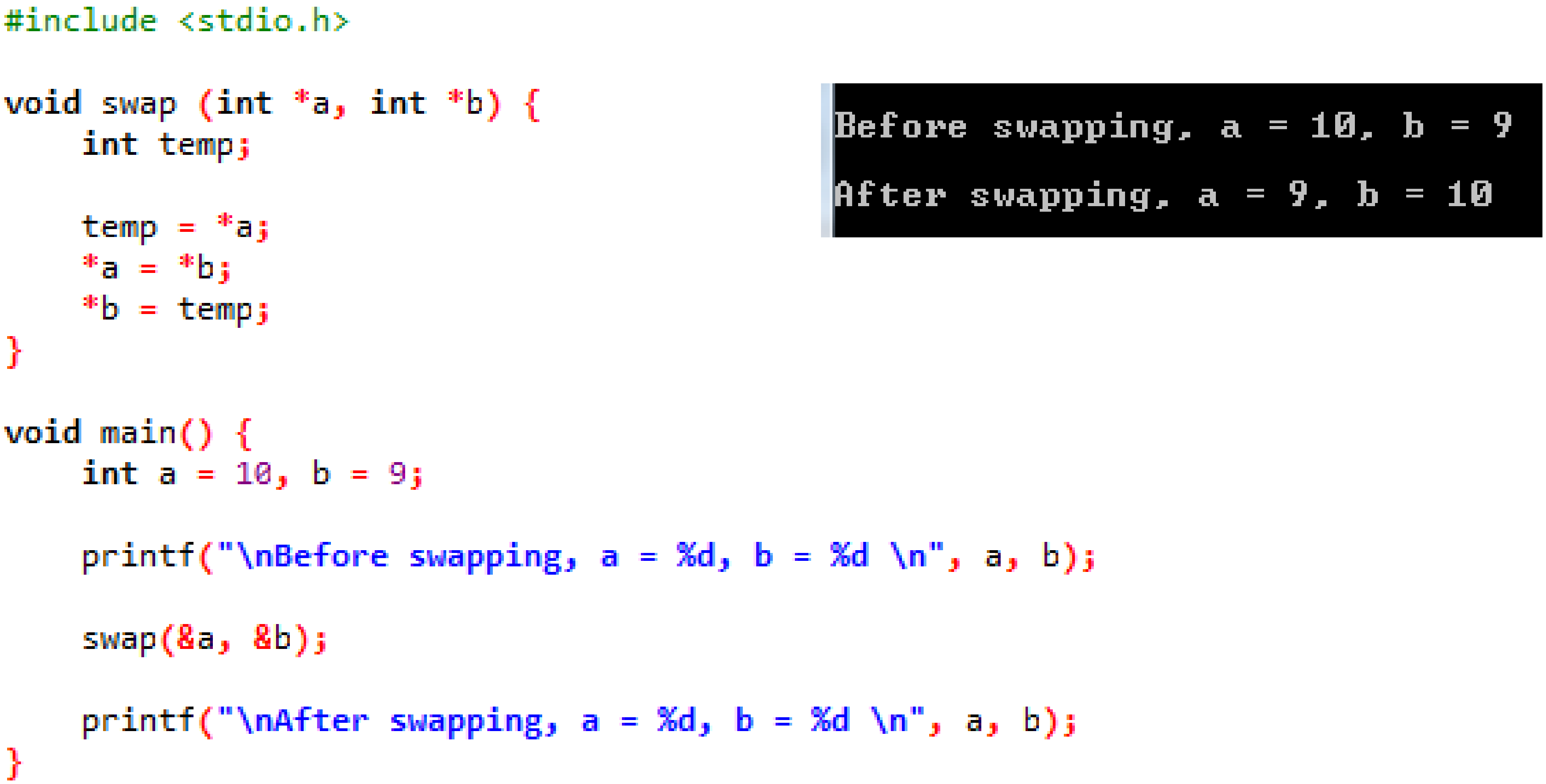
b=9 in the main() function



Stack's values before the callee ends



Function call by reference



```
#include <stdio.h>

void swap (int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

void main() {
    int a = 10, b = 9;

    printf("\nBefore swapping, a = %d, b = %d \n", a, b);

    swap(&a, &b);

    printf("\nAfter swapping, a = %d, b = %d \n", a, b);
}
```

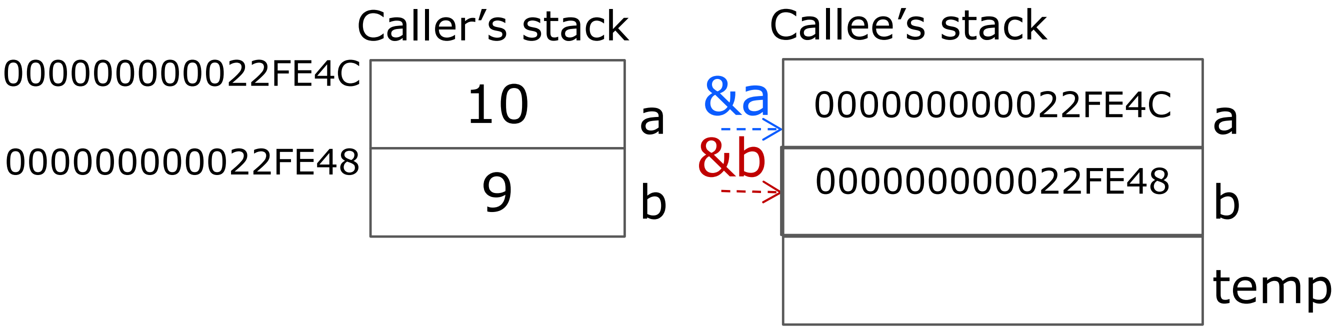
```
Before swapping, a = 10, b = 9

After swapping, a = 9, b = 10
```

- Change on the shared memory can be made by both callee and caller. Pointers are used for reference.

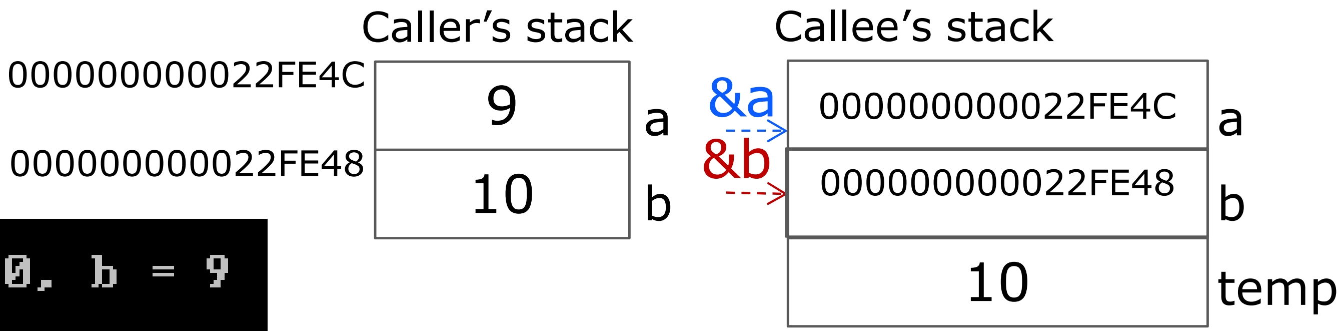
Function call by reference

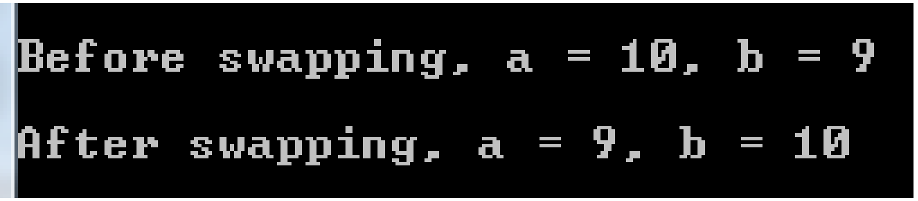
```
#include <stdio.h>

void swap (int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

void main() {
    int a = 10, b = 9;

    printf("\nBefore swapping, a = %d, b = %d \n", a, b);

    swap(&a, &b);

    printf("\nAfter swap
}
```

Stack's values when a=10 and b=9 in the main() function



Stack's values before the callee ends

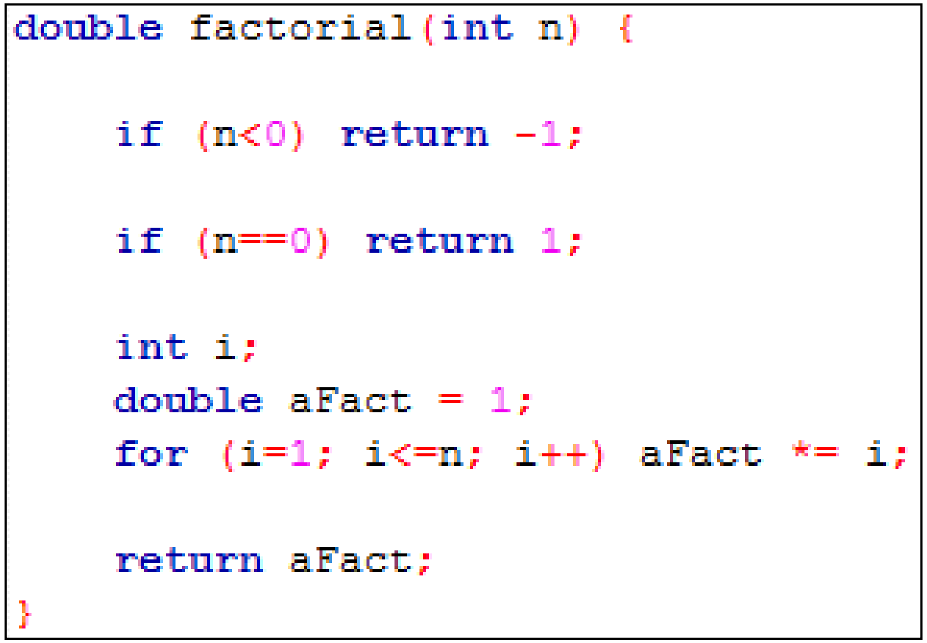




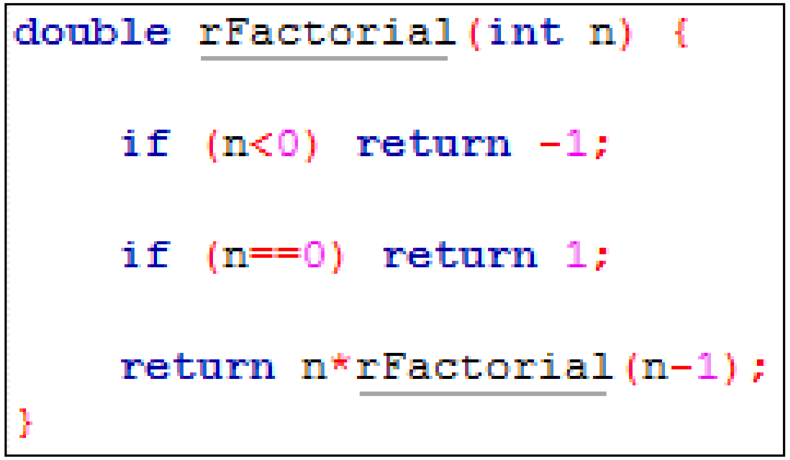
```
Before swapping, a = 10, b = 9
After swapping, a = 9, b = 10
```

Recursion

- A recursive function is a function that calls itself either directly or indirectly.
- When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set.



```
double factorial(int n) {

    if (n<0) return -1;

    if (n==0) return 1;

    int i;
    double aFact = 1;
    for (i=1; i<=n; i++) aFact *= i;

    return aFact;
}
```



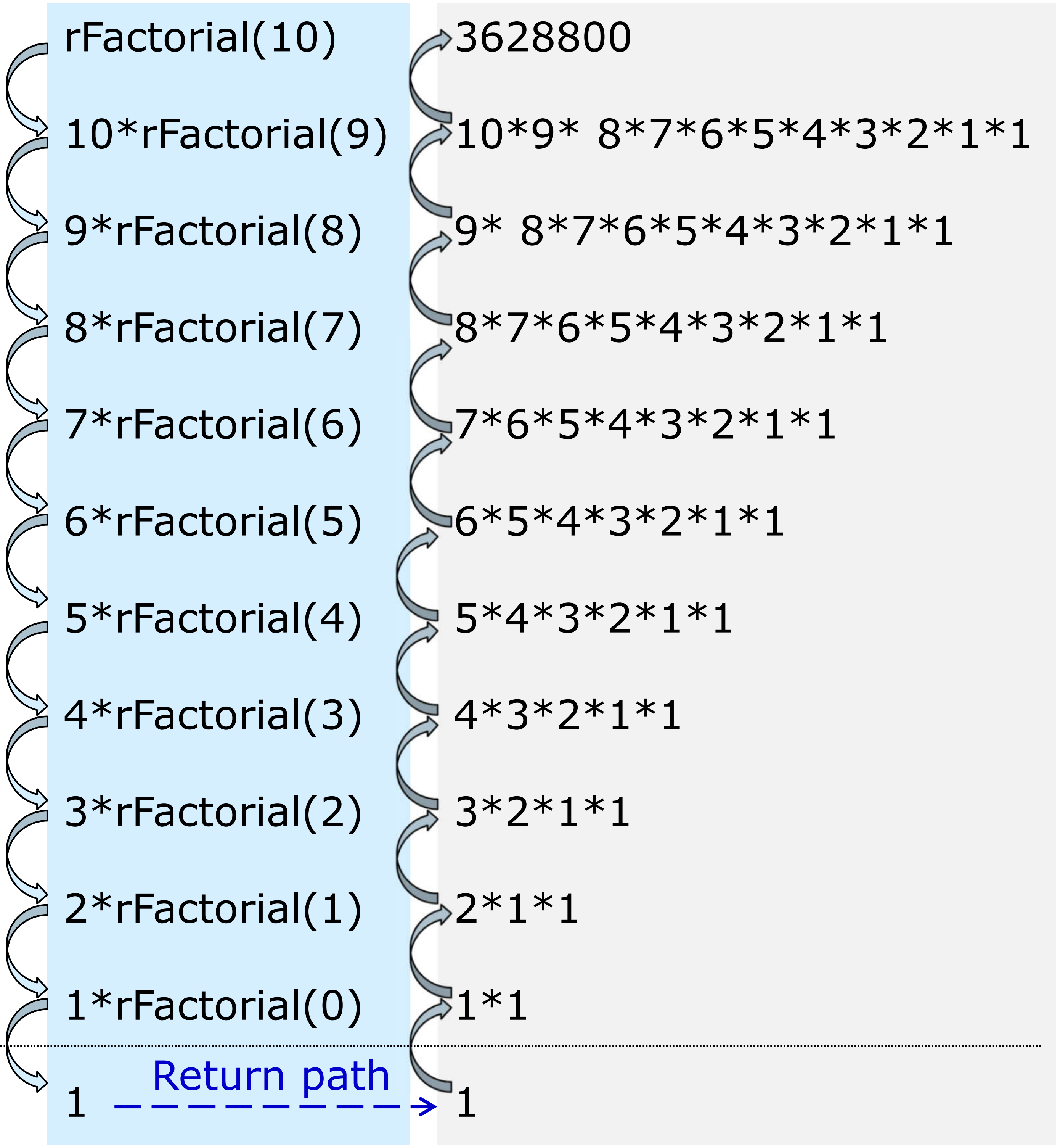
```
double rFactorial(int n) {

    if (n<0) return -1;

    if (n==0) return 1;

    return n*rFactorial(n-1);
}
```

Function to compute the factorial of n:

non-recursive vs. recursive versions

rFactorial(10)

Recursion path



rFactorial(10)	3628800
10*rFactorial(9)	10*9* 8*7*6*5*4*3*2*1*1
9*rFactorial(8)	9* 8*7*6*5*4*3*2*1*1
8*rFactorial(7)	8*7*6*5*4*3*2*1*1
7*rFactorial(6)	7*6*5*4*3*2*1*1
6*rFactorial(5)	6*5*4*3*2*1*1
5*rFactorial(4)	5*4*3*2*1*1
4*rFactorial(3)	4*3*2*1*1
3*rFactorial(2)	3*2*1*1
2*rFactorial(1)	2*1*1
1*rFactorial(0)	1*1
1	1

Recursive cases
with smaller sizes

Base case

Recursion

- Writing a recursive function
  - Determine and write the base cases and their solutions
    - No recursive call is specified for those base cases.
  - Determine and write the recursive (inductive) cases and their solutions
    - Establish a connection between the larger problem and the smaller problems using recursive calls
  - Determine the other cases that are neither base nor recursive cases
    - Check for other constraints with no recursive call

Recursion

- Compute the sum of N first natural numbers
  - sum = 1 + 2 + 3 + … + (N-1) + N
    - Base case: sum(1) = 1
    - Recursive case: sum(N) = sum(N-1) + N
- Compute the factorial of N, a natural number
  - factorial = 1*2*3*…*(N-1)*N
    - Base case: factorial(0) = factorial(1) = 1
    - Recursive case: factorial(N) = factorial(N-1)*N
- Compute the n-th power of x, a floating-point number
  - x^n = x*x*x*…*x
    - Base case: power(x, 0) = 1
    - Recursive case: power(x, n) = power(x, n-1)*x

Recursion

- Estimate a value of e, the natural number

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.71828$$

  - Base case: e(0) = 1
  - Recursive case: e(n) = e(n-1) + 1/factorial(n)

Recursion

- Estimate a value of e^x

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

  - Base case: e_x(0) = 1
  - Recursive case:

    e_x(n) = e_x(n-1) + power(x, n)/factorial(n)

Recursion

- Estimate a value of PI

$$\pi = \sum_{k=0}^{\infty} \frac{(-1)^k 4}{2k+1}$$

  - Base case: pi(0) = 4
  - Recursive case:

    pi(k) = pi(k-1) + power(-1, k)*4/(2*k+1)

Recursion

- Hanoi Tower

Move disks from peg 1 to peg 3 using peg 2 as a temporary holding area in such a way that smaller disks are always on top of larger disks and one disk is moved at a time

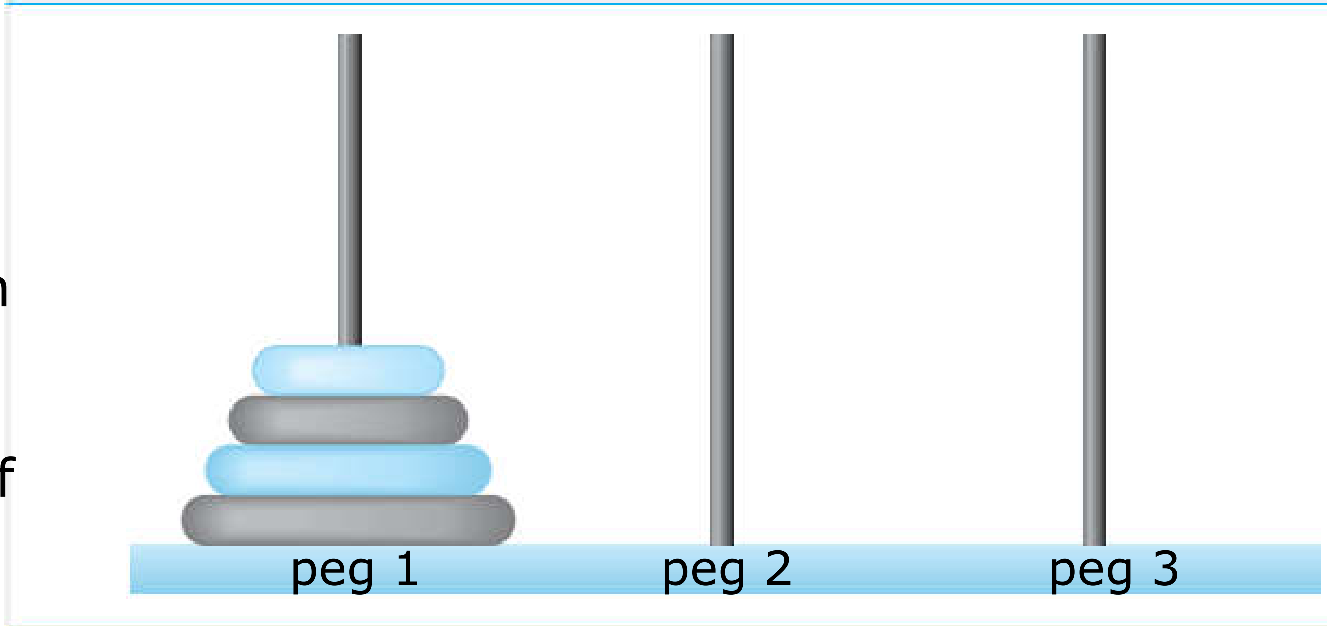


Fig. 5.19 | Towers of Hanoi for the case with four disks.

A recursive function with 4 parameters:

a). The number of disks to be moved

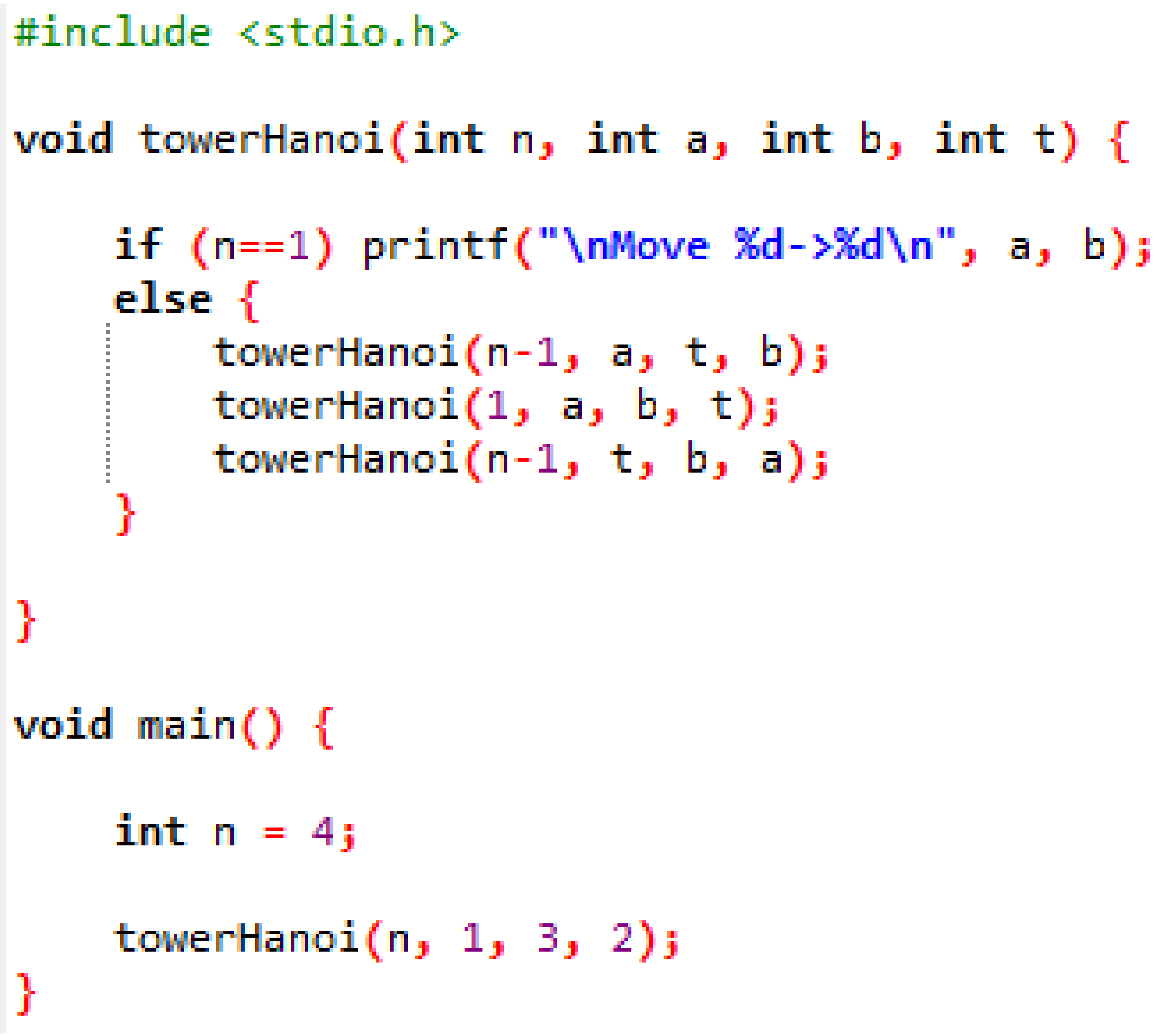
b). The peg on which these disks are initially threaded

c). The peg to which this stack of disks to be moved

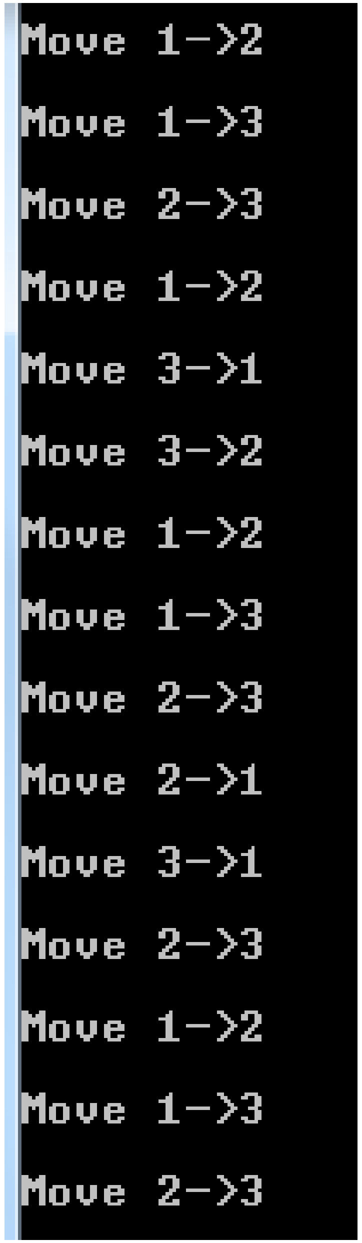
d). The peg to be used as a temporary holding area

Recursion

- Hanoi Tower

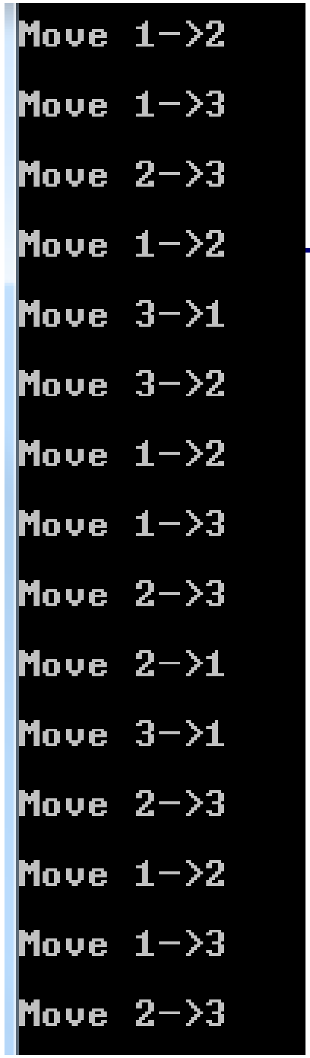


```c
#include <stdio.h>

void towerHanoi(int n, int a, int b, int t) {

    if (n==1) printf("\nMove %d->%d\n", a, b);
    else {
        towerHanoi(n-1, a, t, b);
        towerHanoi(1, a, b, t);
        towerHanoi(n-1, t, b, a);
    }

}

void main() {

    int n = 4;

    towerHanoi(n, 1, 3, 2);
}
```



```
Move 1->2
Move 1->3
Move 2->3
Move 1->2
Move 3->1
Move 3->2
Move 1->2
Move 1->3
Move 2->3
Move 2->1
Move 3->1
Move 2->3
Move 1->2
Move 1->3
Move 2->3
```

Hanoi Tower



```
Move 1->2
Move 1->3
Move 2->3
Move 1->2
Move 3->1
Move 3->2
Move 1->2
Move 1->3
Move 2->3
Move 2->1
Move 3->1
Move 2->3
Move 1->2
Move 1->3
Move 2->3
```

```
towerHanoi(4, 1, 3, 2)
        towerHanoi(3, 1, 2, 3)
                towerHanoi(2, 1, 3, 2)
                        towerHanoi(1, 1, 2, 3)
                        towerHanoi(1, 1, 3, 2)
                        towerHanoi(1, 2, 3, 1)
                towerHanoi(1, 1, 2, 3)
                towerHanoi(2, 3, 2, 1)
                        towerHanoi(1, 3, 1, 2)
                        towerHanoi(1, 3, 2, 1)
                        towerHanoi(1, 1, 2, 3)
        towerHanoi(1, 1, 3, 2)
        towerHanoi(3, 2, 3, 1)
                towerHanoi(2, 2, 1, 3)
                        towerHanoi(1, 2, 3, 1)
                        towerHanoi(1, 2, 1, 3)
                        towerHanoi(1, 3, 1, 2)
                towerHanoi(1, 2, 3, 1)
                towerHanoi(2, 1, 3, 2)
                        towerHanoi(1, 1, 2, 3)
                        towerHanoi(1, 1, 3, 2)
                        towerHanoi(1, 2, 3, 1)
```

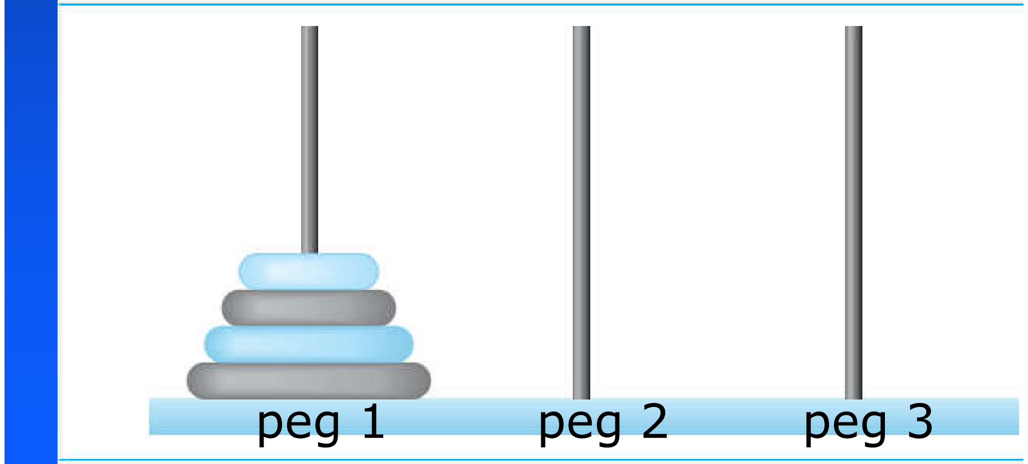


Fig. 5.19 | Towers of Hanoi for the case with four disks.

Recursion

- Recursive function’s concern
  - No saving in storage
  - Not faster
  - Infinite recursion
    - No base case is defined or base case can not be reached.
- Recursive function’s advantages
  - Compact recursive code
  - Easy to write and understand
  - Convenient for recursively defined data structures and problems

Summary

- A function is a processing unit for a <u>specific task</u>.
  - Divide-and-conquer approach
  - Reusability
  - Information hiding
  - Abstraction
  - Support for debugging

→ Manageable program development

Summary

- Function definition
- Function declaration
- Function call
  - By value
  - By reference with pointer implementation
- Recursion
  - Recursive problem
  - Recursive data structure

Chapter 6: Functions