Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2022

Functions

Functions

- The heart of effective problem solving is problem decomposition
  - Breaking a problem into small, manageable pieces
  - In C, the function construct is used to implement this "top-down" method of programming

- A program consists of one or more files
- Each file contains zero or more functions
- One of functions is a `main()` function
- Program execution begins with `main()`, which can call other functions

Function Definition (1)

```
type function_name(parameter_list)        /* function header */
{
    declarations                           /* function body */
    statements
}
```

- A function definition starts with the *type* of the function
  - If no value is returned, then the *type* is `void`
  - If NOT `void`, then the value returned by the function will be converted, if necessary, to this *type*
- Parameter list
  - A comma-separated list of declarations
  - Formal parameters of the function

Function Definition (2)

```c
#include <stdio.h>
#include <assert.h>

int fact(int n) {
    int i, product = 1;

    for (i = 2; i <= n; i++) product *= i;
    return product;
}

int main(void) {
    int n, m, comb;

    scanf("%d %d", &n, &m);
    assert(n >= 0 && m >= 0);
    assert(n >= m);
    comb = fact(n) / (fact(m) * fact(n-m));
    printf("%dC%d = %d\n", n, m, comb);
    return 0;
}
```

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

Function Definition (3)

```
void nothing(void) { }      /* This function does nothing */

double twice(double x)
{
    return 2.0 * x;
}

/* If a function definition does not specify the
   function type, it is int by default */
all_add(int a, int b)
{
    int c;
    ...
    return (a + b + c);
}
```

Why Functions?

- *Why write programs as collections of many small functions?*
- It is simpler to correctly write a small function to do one job
  - Easier writing and debugging
- It is easier to maintain or modify such a program
- Small functions tend to be self-documenting and highly readable
- Functions can be reused

`return` Statement (1)

- `return;`
  `return` *expression*;

- When a `return` statement is encountered,
  - execution of the function is terminated and
  - control is passed back to the calling environment

  - ```
    return;
    return ++a;
    return a * b;
    ```
    ```

return Statement (2)

```
float f(char a, char b, char c) {
    int i = a + b + c;
    return i;            /* value returned will be converted to a float */
}

double abs_value(double x) {
    if (x >= 0.0) return x;
    else return -x;
}

int main() {
    int c;

    while (...) {
        getchar();      /* Even though a function returns a value, */
                        /* a program does not need to use it       */
        c = getchar();
        ...
    }
}
```

Function Prototypes

- Functions should be declared before they are used
- Function prototype:

```
type function_name(parameter_type_list)
```

(e.g.) `double sqrt(double);`

- tells the compiler the number and types of arguments passed to the function
- tells the type of the value returned by the function
- allows the compiler to check the code more thoroughly
- Identifiers are optional
  `void f(char c, int i);` ⇔ `void f(char, int);`

Styles for Function Definition Order (1)

- `#include` and `#define` at the top of file
- `typedef`
- Enumeration types, structures, and unions
- A list of function prototypes
- Function definitions, starting with `main()`

Styles for Function Definition Order (2)

```c
#include <stdio.h>
#define N       7

void prn_header(void);
long power(int, int);
void prn_tbl_of_powers(int);

int main(void) {
    prn_header();
    prn_tbl_of_powers(N);
    return 0;
}
void prn_header(void) {
    ...
}
long power(int m, int n) {
    ...
}
void prn_tbl_powers(int n) {
    ...
    printf("%ld", power(i, j););
    ...
}
```

```c
#include <stdio.h>
#define N       7

void prn_header(void) {
    /* ... */
}
long power(int m, int n) {
    /* ... */
}
void prn_tbl_powers(int n) {
    int i, j;
    /* ... */
    printf("%ld", power(i, j););
    /* ... */
}
int main(void) {
    prn_header();
    prn_tbl_of_powers(N);
    return 0;
}
```

Call-by-Value (1)

- When program control encounters a function name,
  - the function is called, or invoked: the program control passes to that function
  - After the function does its work, the program control is passed back to the calling environment

- Functions are invoked
  - by writing their name and a list of arguments within ( )

- All arguments for a function are passed "call-by-value"
  - Each argument is evaluated, and its value is used locally
  - The stored value of that variable in the calling environment will NOT be changed

Call-by-Value (2)

```c
#include <stdio.h>

int compute_sum(int n);      /* fn prototype */

int main(void)
{
    int n = 3, sum;

    printf("%d\n", n);       /* 3 is printed */
    sum = compute_sum(n);    /* 3 is printed */
    printf("%d\n", n);       /* 6 is printed */
    printf("%d\n", sum);
    return 0;
}
```

argument (pointing to `n` in `compute_sum(n)`)

n, sum

parameter (pointing to `n` in `int compute_sum(int n)`)

```c
int compute_sum(int n)
{
    int sum = 0;

    while (n)
    {
        sum += n;
        n--;
    }
    return sum;
}
```

n, sum

Developing a Large Program (1)

- A large program is typically written in a collection of `.h` and `.c` files

pgm.h

```c
#include <stdio.h>
#include <stdlib.h>

#define     N       3

void fct1(int k);
void fct2(void);
void wrt_info(char *);
```

main.c

```c
#include "pgm.h"

int main(void) {
    char    ans;
    int     i, n = N;

    printf("Do you need any help? ");
    scanf("%c", &ans);
    if (ans == 'y' || ans == 'Y')
        wrt_info("pgm");
    for (i = 0; i < n; i++)
        fct1(i);
    return 0;
}
```

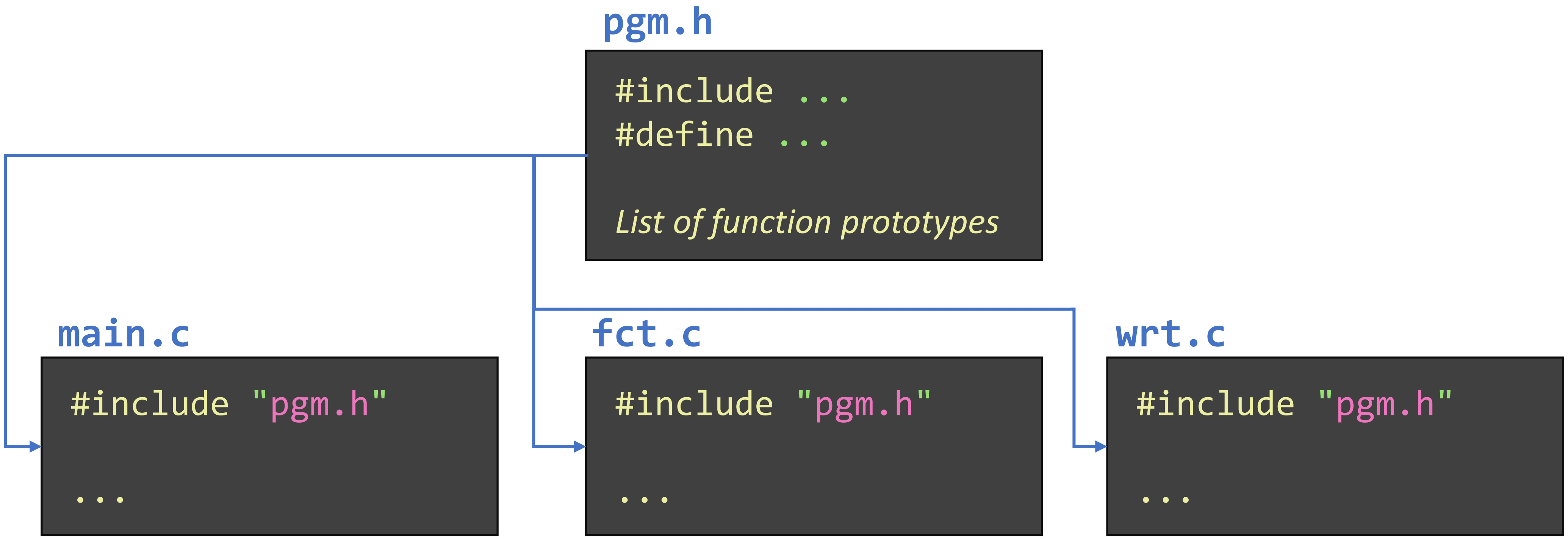
fct.c

```c
#include "pgm.h"

void fct1(int n) {
    int i;

    printf("Hello from fct1()\n");
    for (i = 0; i < n; i++)
        fct2();
}

void fct2(void) {
    printf("Hello from fct2()\n");
}
```

wrt.c

```c
#include "pgm.h"

void wrt_info(char *pgm_name) {
    printf("Usage: %s\n\n", pgm_name);
    printf("Help messages go here...");
}
```

Developing a Large Program (2)



- Because `pgm.h` occurs at the top of each `.c` file, it acts as the "glue" that binds our program together

```
$ cc -o pgm main.c fct.c wrt.c
```

Storage Classes

- Every variable and function in C have two attributes:
  type and *storage class*

- Four storage classes
  - `auto`
  - `register`
  - `extern`
  - `static`

Storage Class auto

- The most common storage class for variable
- Variables declared within function bodies are automatic by default
  - When a block is entered, the system allocates memory for the automatic variables
  - These variables are "local" to the block
  - When the block is exited, the memory is automatically released (the value is lost)

```
void f(int m)
{
    int a, b, c;
    float f;

    ...
}
```

Storage Class `register`

- Tells the compiler that the associated variable should be stored in high-speed registers
- Aims to improve execution speed
  - Declare variables most frequently accessed as `register`

```
int main()
{
    register int i;
    for (i = 0; i < 10; i++)
    {
        ...
    }
    /* block exit will free the register */
}
```

Storage Class `extern` (1)

- One method of transmitting information across blocks and functions is to use external variables
- When a variable is declared outside a function,
  - Storage is permanently assigned to it
  - Its storage class is `extern`
  - The variable is "global" to all functions declared after it

- Information can be passed into a function two ways
  - By use of external variables
  - By use of the parameter mechanism

Storage Class **extern** (2)

```
#include <stdio.h>

int a = 1, b = 2, c = 3;                /* global variables */
int f(void);

int main(void)
{
    printf("%d\n", f());                /* 12 printed */
    printf("%d %d %d\n", a, b, c);      /* 4 2 3 printed */
    return 0;
}
int f(void)
{
    int b, c;                           /* b and c are local */

    a = b = c = 4;                      /* global b, c are masked */
    return (a + b + c);
}
```

Storage Class extern (3)

main.c

```
#include <stdio.h>
int a = 1, b = 2, c = 3;                    /* global variables */
int f(void);

int main(void)
{
    printf("%d\n", f());                    /* 12 printed */
    printf("%d %d %d\n", a, b, c);          /* 4 2 3 printed */
    return 0;
}
```

f.c

```
int f(void)
{
    extern int a;                           /* look for it elsewhere */
    int b, c;

    a = b = c = 4;
    return (a + b + c);
}
```

The keyword extern is used to tell the compiler to "look for elsewhere, either in this file or in some other file."

Storage Class `static`

- Allows a local variable to retain its previous value when the block is reentered
- In contrast to ordinary `auto` variables

```
void f(void)
{
    static int cnt = 0;

    cnt++;
    if (cnt % 2 == 0)
        ...
    else
        ...
}
```

- The first time the function `f()` is invoked `cnt` is initialized to zero
- On function exit, `cnt` is preserved in memory
- Whenever `f()` is invoked again, `cnt` is not reinitialized

Default Initialization

- `external` and `static` variables
  - Initialized to zero by the system, if not explicitly initialized by the programmer

- `auto` and `register` variables
  - Usually not initialized by the system
  - Have "garbage" values

Block Scope Rules (1)

- Basic rules of scoping
  - Identifiers are accessible only within the block in which they are declared
  - They are unknown outside the boundaries of that block

```
{
    int a = 2;               /* outer block a */
    printf("%d\n", a);       /* 2 printed */
    {
        int a = 5;           /* inner block a */
        printf("%d\n", a);   /* 5 printed */
    }
    printf("%d\n", ++a);     /* 3 printed */
}
```

- Nested blocks
  - An outer block name is valid unless an inner block redefines it
  - If redefined, the outer block name is hidden, or masked, from the inner block

```
{
    int a_outer = 2;
    printf("%d\n", a_outer);
    {
        int a_inner = 5;
        printf("%d\n", a_inner);
    }
    printf("%d\n", ++a_outer);
}
```

Block Scope Rules (2)

```
int main(void)
{
    int a = 1, b = 2, c = 3;
    printf("%d %d %d\n", a, b, c);               /* 1 2 3 */
    {
        int b = 4;
        float c = 5.0;
        printf("%d %d %.1f\n", a, b, c);         /* 1 4 5.0 */
        a = b;
        {
            int c;
            c = b;
            printf("%d %d %d\n", a, b, c);       /* 4 4 4 */
        }
        printf("%d %d %.1f\n", a, b, c);         /* 4 4 5.0 */
    }
    printf("%d %d %d\n", a, b, c);               /* 4 2 3 */
}
```

Block Scope Rules (3)

- Parallel blocks
  - Two blocks can come one after another
  - The 2nd block has no knowledge of the variables declared in the 1st block

- Why blocks?
  - To allow memory for variables to be allocated where needed
  - Block exit releases the allocated storage

```
{
    int a, b;

    ...
    {                   /* inner block 1 */
        float b;        /* int a is known, but not int b */
        ...
    }
    ...
    {                   /* inner block 2 */
        float a;        /* int b is known, but not int a */
        ...             /* nothing in inner block 1 is known */
    }
}
```

Declaration vs. Definition

- Declaration
  - Variable declaration: specifies the variable name and its type
  - Function declaration: specifies the function name, the number and type of arguments and its return type
  - A variable or a function can be declared any number of times

```
extern int count;
extern int calc(int, int);
double f(double, double);
```

- Definition
  - A declaration that also causes memory to be reserved for the variable or function
  - A variable or a function can be defined only once

```
int count;
int calc(int a, int b)
{
    static int cnt = 0;

    cnt++;
    return (a + b);
}
```

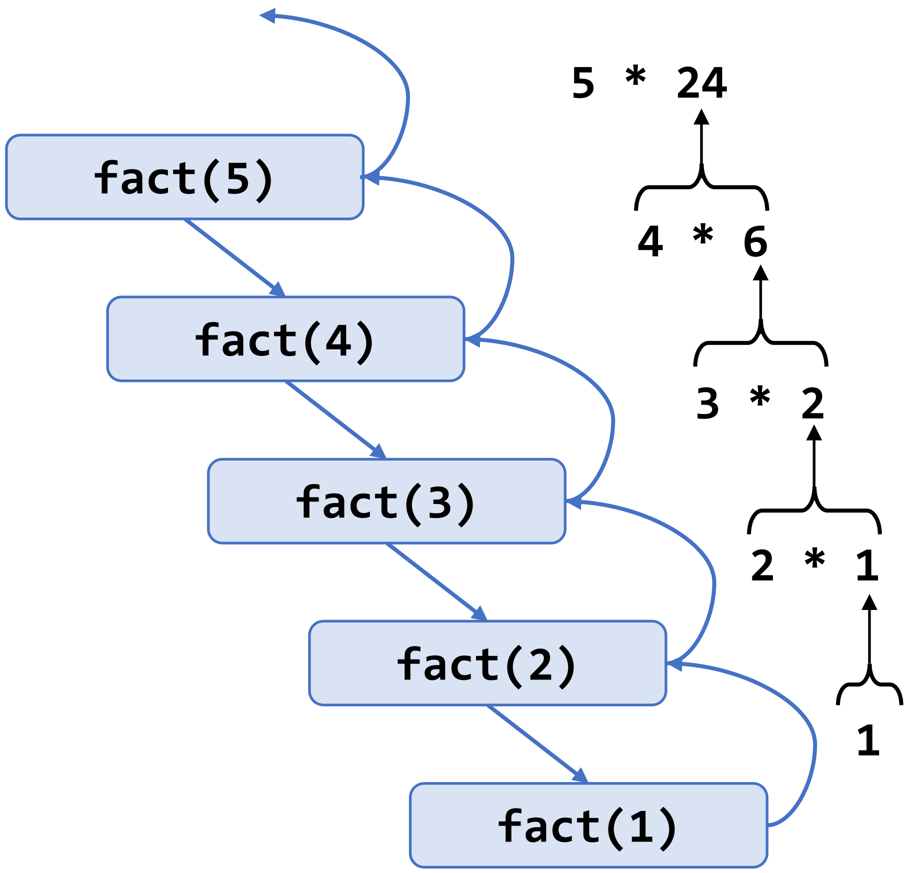
Lifetime vs. Visibility

scope	Type	Storage Class	Lifetime	Visibility
Block	**Variables**	`auto`	Block start ~ end	Within the block
		`register`	Block start ~ end	Within the block
		`static`	Program start ~ end	Within the block
		`extern`	Program start ~ end	Within the block
File	**Variables**	`extern`	Program start ~ end	Remainder of source file
		`static`	Program start ~ end	Remainder of source file (single source file only)
	Functions	`extern`	Program start ~ end	Remainder of source file
		`static`	Program start ~ end	Remainder of source file (single source file only)

Recursion

- A function is recursive if it calls itself, either directly or indirectly

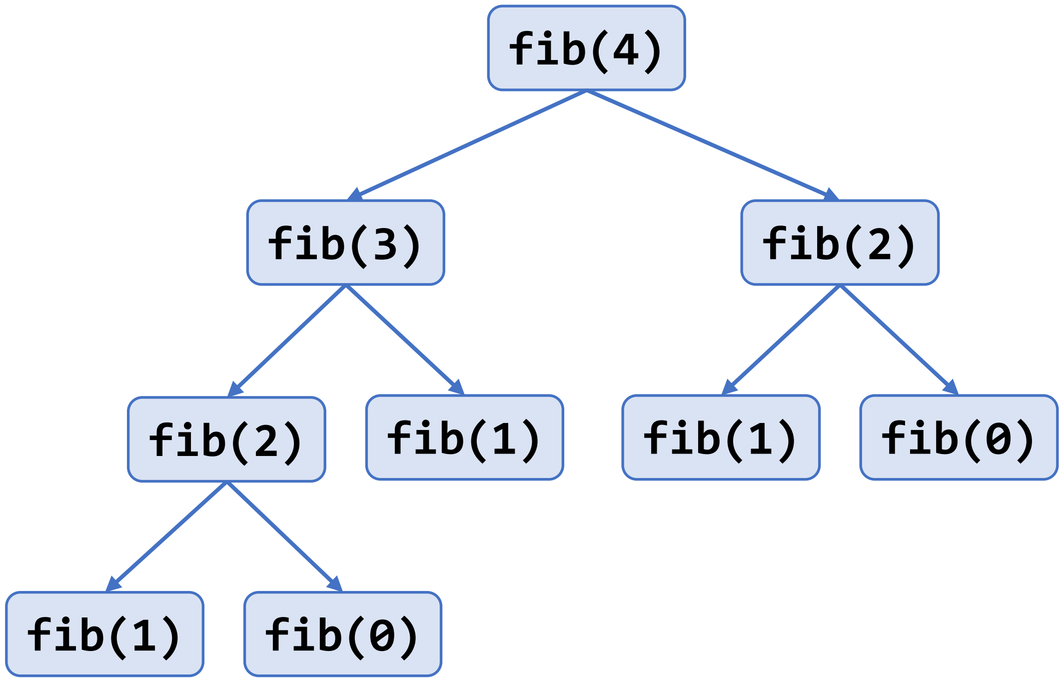
```c
#include <stdio.h>

int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}

int main(void)
{
    printf("%d! = %d\n", 5, fact(5));
    return 0;
}
```



Fibonacci Sequence

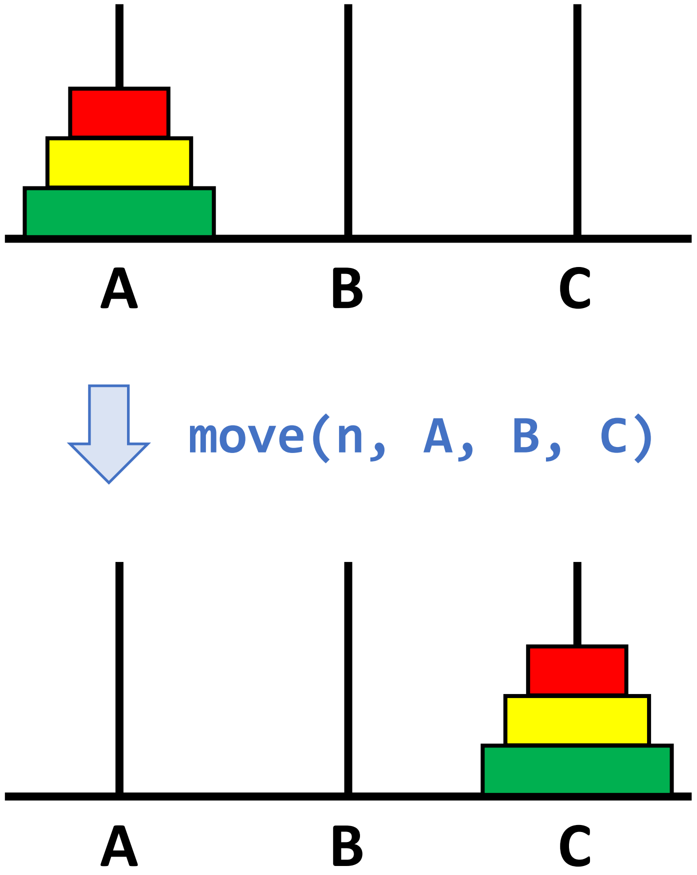
- $f_0 = 0, \;\; f_1 = 1, \;\; f_n = f_{n-1} + f_{n-2} \; (n \; \geq 2)$

```c
#include <stdio.h>

int fib(int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}

int main(void)
{
    int i;

    for (i = 0; i <= 10; i++)
        printf("fib(%d) = %d\n", i, fib(i));
}
```

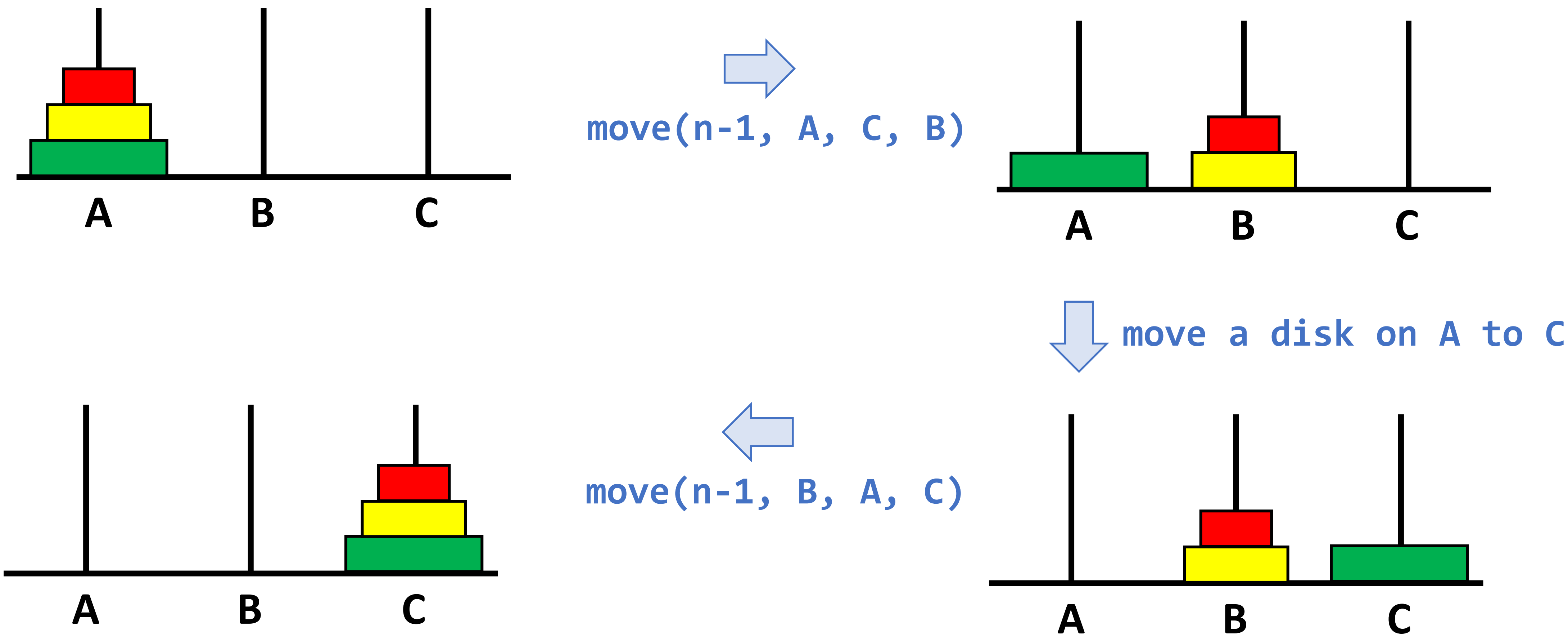


Tower of Hanoi (1)

- Tower of Hanoi game
  - There are three towers labeled A, B, and C
  - The game starts with *n* disks
  - The object of the game is to move all disks on tower A to tower C
  - Restriction: a larger disk cannot be placed on a smaller disk
  - Task of transferring the *n* disks from tower A to tower C



Tower of Hanoi (2)



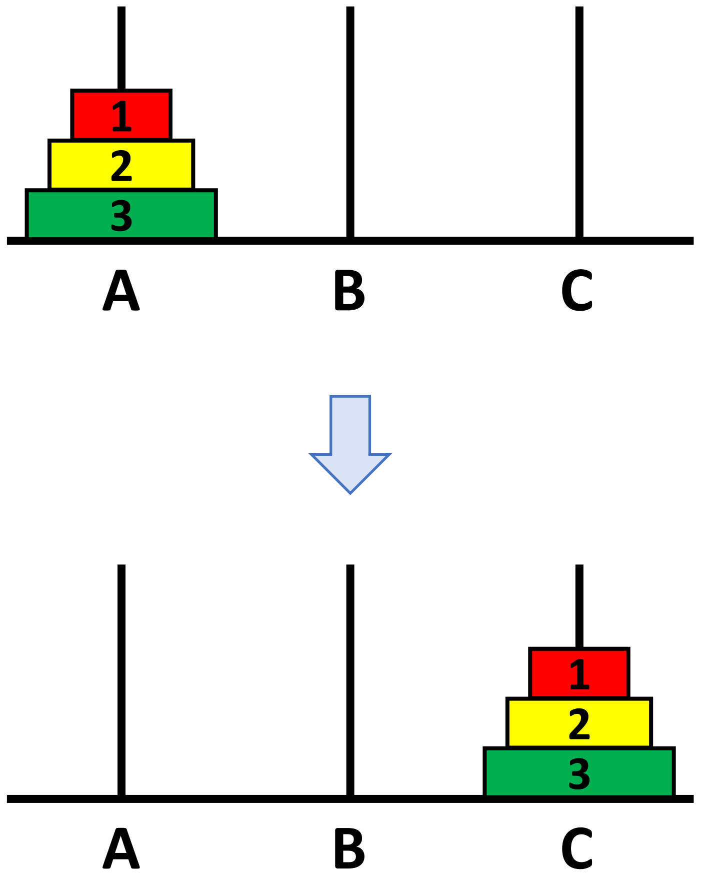
Tower of Hanoi (3)

```c
#include <assert.h>
#include <stdio.h>

int step = 0;

void move(int, char, char, char);

int main(void)
{
    int n;
    printf("Input n (>0): ");
    scanf("%d", &n);
    assert(n > 0);
    move(n, 'A', 'B', 'C');
    return 0;
}
```

```c
void move(int n, char src, char spare, char dest)
{
    if (n == 1)
    {
        step++;
        printf("%5d: Move disk %d from tower %c to %c\n",
            step, 1, src, dest);
    }
    else
    {
        move(n-1, src, dest, spare);
        step++;
        printf("%5d: Move disk %d from tower %c to %c\n",
            step, n, src, dest);
        move(n-1, spare, src, dest);
    }
}
```

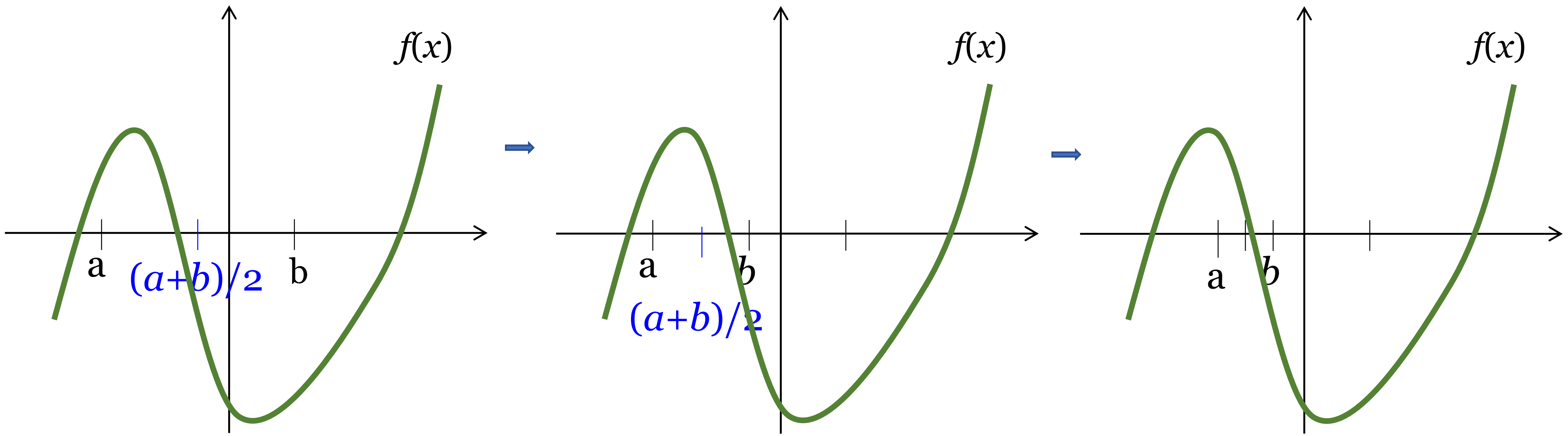
Tower of Hanoi (4)

```
Input n (>0): 3
1: Move disk 1 from tower A to C
2: Move disk 2 from tower A to B
3: Move disk 1 from tower C to B
4: Move disk 3 from tower A to C
5: Move disk 1 from tower B to A
6: Move disk 2 from tower B to C
7: Move disk 1 from tower A to C
```

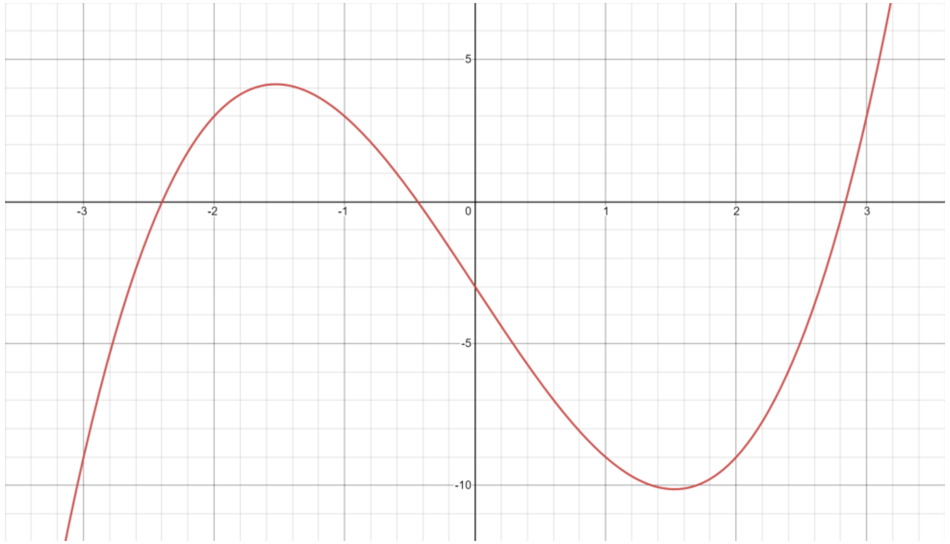


Bisection (1)

- Finding a root of a function
  - For a continuous function $f(x)$, when $f(a) * f(b) \leq 0$, there is at least one root in $[a, b]$



Bisection (2)




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

int cnt = 0;
const double eps = 1e-13;
double f(double x);

double bisection(double a, double b)
{
    double m = (a + b) / 2.0;

    cnt++;
    if (f(m) == 0.0 || (b - a) < eps)
        return m;
    else if (f(a)*f(m) < 0.0)
        return bisection(a, m);
    else
        return bisection(m, b);
}
```

```c
double f(double x)
{
    return pow(x, 3) - 7.0*x - 3.0;
}

int main(void)
{
    double a = -10.0, b = 10.0;
    double root;

    assert(f(a)*f(b) <= 0.0);
    root = bisection(a, b);
    printf("No. of calls: %d\n", cnt);
    printf("root: %f, f(root): %f\n", root, f(root));
}
```

Mathematical Functions

- `#include <math.h>` (Also add "`-lm`" to link with the math library)

Defined function prototype	Function call	Meaning
`double cos(double x);`	cos(*expr*)	$\cos x$
`double sin(double x);`	sin(*expr*)	$\sin x$
`double tan(double x);`	tan(*expr*)	$\tan x$
`double acos(double x);`	acos(*expr*)	$\mathrm{acos}\, x$
`double asin(double x);`	asin(*expr*)	$\mathrm{asin}\, x$
`double atan(double x);`	atan(*expr*)	$\mathrm{atan}\, x$
`double exp(double x);`	exp(*expr*)	e^x
`double log(double x);`	log(*expr*)	$\log_e x$
`double log10(double x);`	log10(*expr*)	$\log_{10} x$
`double ceil(double x);`	ceil(*expr*)	$\lceil x \rceil$ (the smallest integer not less than x)
`double floor(double x);`	floor(*expr*)	$\lfloor x \rfloor$ (the largest integer not greater than x)
`double fabs(double x);`	fabs(*expr*)	$\lvert x \rvert$
`double fmod(double x, double y);`	fmod(*expr1*, *expr2*)	$x\ (mod\ y)$
`double pow(double x, double y);`	pow(*expr1*, *expr2*)	x^y
`double sqrt(double x);`	sqrt(*expr*)	$\sqrt{x}$

Using Assertions

- `assert`(*expr*)
  - In the standard header file `assert.h`
  - If *expr* is false, the system will print a message, and the program will be aborted
  - This can be used to ensure that the value of expression is what you expect it to be
  - Add robustness to the code

```
int f(int a, int b)
{
    assert(a==1 || a==-1);      /* the value of a should be either 1 or -1 */
    assert(b>=7 && b<=11);      /* the value of b should be in [7, 11] */
    ...
}
```

```
Enter two numbers: 1 1
a.out: assert.c:6: f: Assertion `b>=7 && b<=11' failed.
Aborted (core dumped)
```

printf()

- `printf`(*format_string*, *other_arguments*)
  - In the standard header file `stdio.h`
  - (e.g.) `printf("she sell %d %s for %f", 99, "sea shells", 3.77);`

    conversion spec.
  - Conversion specification
    - How the corresponding argument is printed
    - Begins with % and ends with a conversion character
  - Conversion character

c	as a character
d	as a decimal integer
u	as an unsigned decimal integer
o	as an unsigned octal integer
x, X	as an unsigned hexadecimal integer

e	as a floating-point number (e.g., 7.123000e+00)
E	as a floating-point number (e.g., 7.123000E+00)
f	as a floating-point number (e.g., 7.123000)
s	as a string

printf()

Assume: `int i = 123; double x = 28.123456789;`
`char c = 'A', str[] = "Blue moon!"`

Format	Argument	How it is printed	Remarks
`%d`	`i`	`"123"`	field width 3 by default (minimum field width)
`%05d`	`i`	`"00123"`	padded with zeros, field width 5
`%7o`	`i`	`"    173"`	field width 7, right adjusted (default), octal
`%-9x`	`i`	`"7b       "`	left adjusted, hexadecimal
`%-#9x`	`i`	`"0x7b     "`	left adjusted, 0x prepended, hexadecimal
`%f`	`x`	`"28.123457"`	six digits at the right of the decimal point by default
`%11.5f`	`x`	`"   28.12346"`	field width 11, precision 5
`%-14.5e`	`x`	`"2.81235e+01   "`	field width 14, precision 5, left adjusted, e-format
`%c`	`c`	`"A"`	field width 1 by default (one character)
`%2c`	`c`	`" A"`	field width 2, right adjusted
`%-3c`	`c`	`"A  "`	field width 3, left adjusted
`%s`	`str`	`"Blue moon!"`	field width 10 by default (the number of chars in the string)
`%3s`	`str`	`"Blue moon!"`	If the specified field width is too short, the field width becomes default
`%.6s`	`str`	`"Blue m"`	precision 6 (the maximum number of characters to be printed)
`%-11.8s`	`str`	`"Blue moo   "`	precision 8, field width 11, left adjusted

scanf()

- `scanf`(*format_string*, *other_arguments*)
  - In the standard header file `stdio.h`

```
char    a, b, c, s[100];
int     n;
double  x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

  - Conversion character

c	a character, including white space	**f**	a floating-point number (float)
d	a decimal integer (int)	**lf**	a floating-point number (double)
ld	a decimal integer (long)	**s**	a string