Functions

Fundamentals of Computer and Programming

Fall 2020

Bahador Bakhshi

CE & IT Department, Amirkabir University of Technology

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

Introduction

- Until now, we learned to develop simple algorithms
  - Interactions, Mathematics, Decisions, and Loops
- Real problems: very complex
  - Compressing a file
  - Calculator
  - Games, MS Word, Firefox, IDE, Compiler, …
- Cannot be developed at once
  - Divide the problem into smaller sub-problems
  - Solve the sub-problems
  - Put the solutions altogether to get the final solution
- Modular programming

Modular programming

- Solving a large and complex problem
- Design the overall algorithm
- Some portions are black-box
  - We know what each box does
  - But we don't worry how
  - Later, we think about the black-boxes and develop them
- Black-boxes are implemented by functions

Modular programming: Advantages

- Easy to develop and understand
- Reusability
  - Something is used frequently
    - Mathematic: Square, Power, Sin, …
    - Programming: Printing, Reading
  - Develop it one time, use it many times
- Multiple developers can work on different parts
- Each module can be tested and debugged separately

Functions in C

- Functions in mathematics
  - z = f(x,y)
- Functions in C
  - Queries: Return a value
    - **`sin(), fabs()`**
  - Commands: do some tasks, do not return any value
    - **`printf_my_info(...)`**

Functions in C

- Three steps to use functions in C
- Function prototype (declaration) (اعلان تابع) (معرفی الگوی تابع)
  - Introduce the function to compiler
- Function definition (تعریف تابع)
  - What the function does
- Function call (فراخوانی تابع)
  - Use the function

Function prototype

```
<output type> <function name>(<input
  parameter types>);
```

- <output type>
  - Queries: `int, float,…`
  - Command: `void`
- <function name> is an identifier
- <input parameter list>
  - <type>, <type>, …
    - `int, float, …`
  - `void`

Function definition

```
<output type> <function name>(<input parameters>)
{
  <statements>
}
```

- <output type>
  - Queries: `int, float,`…
  - Command: `void`
- <function name> is an identifier
- <input parameters>
  - <type> <identifier>, <type> <identifier>, …
    - `int in, float f,` …
  - `void`
- Function definition should be out of other functions
  - Function in function is not allowed

Function call

- Command function

```
<function name> (inputs);
```

- Query function

```
<variable> = <function name>(inputs);
```

- Inputs should match by function definition
- Functions are called by another function
  - Function call comes inside in a function

Example

```
/* Function declaration */
void my_info(void);

int main(void){
  printf("This is my info");
  my_info(); /* Function call */
  printf("==============");
  return 0;
}

/* Function definition */
void my_info(void){
  printf("Student name is Dennis Ritchie\n");
  printf("Student number: 9822222\n");
```

Function declaration is optional if program is developed in a single file

```
void my_info(void){
  printf("My name is Dennis Ritchie\n");
  printf("My student number: 98222222\n");
}

int main(void){
  my_info();
  printf("---------------\n");
  my_info();
  return 0;
}
```

Function Declaration?!!!!

- Is function declaration needed?
- Is there any useful application of application declaration?
- Yes!
- Libraries are implemented using it
  - .h files contains the function declarations
    - and also other definitions
  - .so, .a, .dll, … are the compiled function definitions

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

Input Parameters

- Inputs of function
  - No input: **void**
  - One or multiple inputs
- Each input should have a type
- Input parameters are split by “,”

```
void f(void)
void f(int a)
void f(int a, float b)
void f(int a, b) //compile error
```

Example: print_sub function

تابعی که دو عدد را بگیرد
و تفاضل آنها را چاپ کند.

```
#include <stdio.h>

void print_sub(double a, double b){
  double res;
  res = a - b;
  printf("Sub of %f and %f is %f\n", a, b, res);
}

int main(void){
  double d1 = 10, d2 = 20;
  print_sub(56.0, 6.0);    //What is the output?
  print_sub(d1, d2);       //output?
  print_sub(d1, d2 + d2); //output?
  return 0;
}
```

How Does Function Call Work?

- Function call is implemented by "stack"
- Stack is a logical part of the main memory
- Variables of function and its input variables are in stack
- When a function calls
  - Its variables including the inputs are allocated in stack
  - The value of input parameters from caller function is pushed to stack of called function
    - They are *copied* in to the variables of function
- When function finished, its stack is freed

print_sub: What happen?

```
print_sub(56.0, 6.0);
```

- 56.0 is copied the memory location **a**
- 6.0 is copied to memory location **b**

```
double a = 56.0;

double b = 6.0;

double res;

res = a - b;
```

print_sub: What happen?

```
print_sub(d1, d2);
```

- Value of **d1** is copied the memory location **a**
- Value of **d2** is copied to memory location **b**

```
double a = 10.1;
double b = 20.2;
double res;
res = a - b;
```

Call by Value

Call by value

- In call by value mechanism
  - The values are copied to the function
- If we change values in the function
  - The copied version is changed
  - The original value does not affected
- Call by value inputs cannot be used to produce output

add function (wrong version)

```
void add(double a, double b, double res){
  res = a + b;
  return;
}

int main(void){
  double d1 = 10.1, d2 = 20.2;
  double result = 0;
  add(56.0, 6.7, result);
  printf("result = %f\n", result);   result = 0
  add(d1, d2, result);
  printf("result = %f\n", result);   result = 0
}
```

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

Producing output

- What we have seen are the “Command”
- Query functions
  - Produce output
  - Output cannot be produced by the “call by value” parameters
- To produce an output
  - Declare output type
  - Generate the output by **`return`**

The `return` command

- To generate a result by a function

```
return <value>;
```

- Only one value can be returned
- **`return`** finishes running the function
- Function can have multiple return
  - Only one of them runs each time
- The type of the returned value = the result type
  - Otherwise, cast

Exmaple: my_fabs (Version 1)

```
double my_fabs(double x){
   double res;
   if(x >= 0)
      res = x;
   else
      res = -1 * x;
   return res;
}

void main(void){
   double d = -10;
   double b;
   b = my_fabs(d);
   printf("%f\n", b);                          10
   printf("%f\n", my_fabs(-2 * b));            20
}
```

Exmaple: my_fabs (Version 2)

```
double my_fabs(double x){
  if(x >= 0)
     return x;
  return (-1 * x);
}

void main(void){
  double d = -10;
  double b;
  b = my_fabs(d);
  printf("b  = %f\n", b);
  b = my_fabs(-2 * d);
  printf("b  = %f\n", b);
}
```

Output of functions

- A function can produce at most one output
- Output of functions can be dropped

```
double f;
sin(f);  //we drop the output of sin
gcd(10, 20);
     //we drop the output of gcd
```

Casting in functions

- Cast for input
  - Prototype: `void f(int a, double b);`
  - Call: `f(10.1, 20.2);`
- Cast for output
  - Prototype: `int f(int a);`
  - Call: `double d = f(10);`
  - Cast in return

```
int f(int a){
   ...
   return 10.20
}
```

Be careful: empty input/output type

- If output or input type is not specified → int
  - Casting may not work

```
f1(a){
    printf("a = %d\n", a);  return a / 2;
}
f2(int a){
    printf("a = %d\n", a);  return a / 2;
}
f3(float a){
    printf("a = %f\n", a);  return a / 2;
}
int main(){
    printf("%d\n", f1(10.5));
    printf("%d\n", f2(10.5));
    printf("%d\n", f3(10.5));
    return 0;
```

Inline Functions & Macro

- Function call using stack has its overhead
  - 2 approaches to reduce the overhead
- **`inline`** function
  - To ask from compiler to compile it as inline, but no guarantee

```
inline int f(float x)
```

- Macros

```
#define PRINT_INT(X) printf("%d\n", X)
```

Example: GCD (بزرگترین مقسوم علیه مشترک)

```
#define PRINT_INT(x) printf("%d\n",x); \
                     printf("====================\n");
inline int gcd(int a, int b){ /* return gcd of a and b */
   int temp;
   while(b != 0){
       temp = a % b;
       a = b;
       b = temp;
   }
   return a;
}

void main(void){
   int i = 20, j = 35, g;
   g = gcd(i, j);
   printf("GCD of %d and %d = ", i , j);
   PRINT_INT(g);
   g = gcd(j, i);
   printf("GCD of %d and %d = ", j , i);
   PRINT_INT(g);
}
```

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

Scope of Variables

- Variables
  - Are declared in the start of functions
  - Are used any where in the function after declaration
  - Cannot be used outside of function
  - Cannot be used in other functions
- Scope of variable
  - A range of code that the variable can be used
- Variable cannot not be used outside of its scope
  - Compile error

Scopes and Blocks

- Scopes are determined by Blocks
  - Start with { and finished by }
  - Example: statements of a function, statement of a **`if`** or **`while`**, …
- Variables
  - Can be declared in a block
  - Can be used in the declared block
  - Cannot be used outside the declared block
- The declared block is the scope of the variable

Variables in Blocks

```
#include <stdio.h>
int main(void){
   int i;
   for(i = 1; i <= 10; i++){
      int number;
      printf("Enter %d-th number: ", i);
      scanf("%d", &number);
      if((number % 2) == 0)
         printf("Your number is even\n");
      else
          printf("Your number is odd\n");
   }
   /* compile error
    printf("The last number is %d\n", number); */
   return 0;
}
```

Nested Scopes/Blocks

- Scopes can be nested
  - Example: Nested **`if`**, nested **`for`**, …

```
void main(){ //block 1
  int i;
  { //block 2
     int j;
     { //block 3
        int k;
     }
     int m;
  }
}
```

Variables in Nested Blocks

- All variables from outer block can be used inner blocks
  - Scope of outer block contains the inner block

- Variables in inner block cannot be used in outer block
  - Scope of the inner block does not contains the outer block

Variables in Nested Blocks: Example

```
int k;
for(int i = 0; i < 10; i++){
    /* block 1 */
    if(i > 5){
        /* block 2 */
        int j = i;
        ...
    }
    while(k > 10){
        /* block 3 */
        int l = i;
        /* int m = j; compile error */
        ...
    }
    /* k = l; compile error */
}
```

Same Variables in Nested Block

- If a variable in inner block has the same identifier of a variable in outer block
  - The inner variable hides the outer variable
  - Changing inner variables does not change outer variable

```
int j = 20, i = 10;
printf("outer i = %d, %d\n", i, j);
while(…){
  int i = 100;
  j = 200;
  printf("inner i = %d, %d\n", i, j);
  ...
}
printf("outer i = %d, %d\n", i, j);
```

Same Variables in Nested Block

- If a variable in inner block has the same identifier of a variable in outer block
  - The inner variable hides the outer variable
  - Changing inner variables does not change outer variable

```
int j =
printf("                                    );
while(…)
  int i
  j = 20
  printf                                    j);
  ...
}
printf("outer i = %d, %d\n", i, j);
```

Do NOT Use It!!!

Local Variables

- All variables defined in a function are the local variable of the function
- Can ONLY be used in the function, not other functions

```
void func(void){
   int i, j;
   float f;
   /* These are local variables */
}
int main(void){
   i = 10; /* compile error, why? */
   f = 0;  /* compile error, why? */
}
```

Global/External Variables

- Global variables are defined outside of all functions
- Global variables are *initialized* to zero
- Global variables are available to all subsequent functions

```
void f(){
  i = 0; // compile error
}
int i;
void g(){
  int j = i; // g can use i
}
```

Global/External Variables: Example

```
int i, j;
float f;
void func(void){
   printf("i = %d \n", i);
   printf("f = %f \n", f);
   i = 20;
}
void f1(){
   printf("%d", i);
}
int main(void){
   f = 1000;
   func();
   f1();
   return 0;
}
```

```
i = 0
f = 1000
```

Parameter Passing by Global Variables: my_fabs (V.3)

```
double x;

void my_fabs(void){
  x = (x > 0) ? x : -1 * x;
}

void main(void){
  double b, d = -10;
  x = d;
  my_fabs();
  b = x;
  printf("b  = %f\n", b);
}
```

Don’t use this method. Parameters should be passed by input parameter list.

Global variable are used to define (large) variables that are used in many functions

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage class of variables
- Function usage example
- Recursion

Storage Classes

- Storage class
  - How memory is allocated for the variable
  - Until when the variable exists
  - How it is initialized
- Storage classes in C
  - Automatic (اتوماتیک)
  - External (خارجی)
  - Static (ایستا)
  - Register (ثبات)

Storage Classes: Automatic

- All local variables are automatic by default
  - Input parameters of a function
  - Variables defined inside a function/block
- Generated at the start of each run of the block
- Destroyed at the end of each run of the block
- Are not initialized

Storage Classes: External

- All global variables are external by default
  - Are initialized by 0
  - Are generated when program starts
  - Are destroyed when program finishes
- Usage of keyword "**`extern`**"
  - To use global variables in other files
  - To use global variables before definition
  - To emphasize that variable is global
    - This usage is optional

Storage Classes: Static

- Keyword "**`static`**" comes before them
- For local variables:
- 1) Generated in the first run of the block
- 2) Destroyed when program finishes
- 3) Initialized
  - If no value → initialized by 0
  - Only initialized in the first run of the block

Storage Classes: Static

- Keyword "**`static`**" comes before them
- For global variables:
- 1) Generated when program starts
- 2) Destroyed when program finishes
- 3) Always initialized
  - If no value → initialized by 0
- *4) Is not accessible for other files*

Storage Classes: Register

- Keyword "**`register`**" comes before them
- Can be used for local variables
- Compiler tries to allocated the variable in registers of CPU
  - But does not guaranteed
  - Registers are very fast and small memories
- Improve performance

Storage Classes, Auto: Examples

```
void f(int i, double d){
  int i2;
  int i3;
  double d2;
  double d3;
  ...
}
```

All variables (i, d, i2, i3, d2, d3) are auto variables

Storage Classes, Extern: Examples

```
int i = 10, j = 20;

void print(void){
  printf("i = %d, j = %d\n", i, j);
}

int main(void){
  extern int i;  // i refers the global i
  int j;         // j is new variable

  print();           i = 10, j = 20
  i = 1000;
  j = 2000;
  print();           i = 1000, j = 20
  return 0;
}
```

Storage Classes: Examples

```
int i;
void func(void){
  int j;
  printf("i = %d \n", i);
  printf("j = %d \n", j);
  i = 20;
}
int main(void){
  func();
  func();
  i = 30;
  func();
  return 0;
}
```

i = 0
j = ???
i = 20
j = ??
i = 30
j = ??

Storage Classes, Static: Examples

```
void func(void){
  int j;
  static int i;
  printf("i = %d \n", i);
  printf("j = %d \n", j);
  i = 20;
}
int main(void){
  func();
  func();
  /* i = 30;   compile error, why? */
  func();
  return 0;
}
```

```
i = 0
j = ???
i = 20
j = ???
i = 20
j = ???
```

Storage Classes, Static: Examples

```
void func(void){
  int j;
  static int i = 10;
  printf("i = %d \n", i);
  printf("j = %d \n", j);
  i = 20;
}

int main(void){
  func();
  func();
  return 0;
}
```

i = 10
j = ???
i = 20
j = ???

Storage Classes, Register: Examples

```
register int i;

for(i = 0; i < 100; i++)
   …
```

Be careful: loop & automatic variables

- According to standard:

"*For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way.*"

- Variable is defined in a block of a loop
- 1) the variable retains its value between iterations of the loop if it is NOT variable length array
- 2) the variable does NOT retain its value between iterations of the loop if it is a variable length array

Loop & Automatic Variables

```
int main(){
    int i;
    for(i = 0; i < 5; i++){
        int j;
        if(i){
            printf("&j = %p, j = %d\n"
                    , &j, j);
            j++;
        }
        else
            j = i;
    }
}
```

&j = 0x7ffd504ab740, j = 0
&j = 0x7ffd504ab740, j = 1
&j = 0x7ffd504ab740, j = 2
&j = 0x7ffd504ab740, j = 3

Loop & Automatic variables

```
int main(){
    int i;
    for(i = 0; i < 5; i++){
        int j[5 * 3 + 1];
        if(i){
            printf("&j[0] = %p, j[0] = %d\n"
                   , &(j[0]), j[0]);
            j[0]++;
        }
        else
            j[0] = i;
    }
}
```

```
&j[0] = 0x7ffdcda151a0, j[0] = 0
&j[0] = 0x7ffdcda151a0, j[0] = 1
&j[0] = 0x7ffdcda151a0, j[0] = 2
&j[0] = 0x7ffdcda151a0, j[0] = 3
```

Loop & Automatic variables

```
int main(){
    int i;
    for(i = 0; i < 5; i++){
        int j[5 * i + 1];
        if(i){
            printf("&j[0] = %p, j[0] = %d\n"
                   , &(j[0]), j[0]);
            j[0]++;
        }
        else
            j[0] = i;
    }
}
```

```
&j[0] = 0x7ffdc879d680, j[0] = 0
&j[0] = 0x7ffdc879d670, j[0] = 0
&j[0] = 0x7ffdc879d660, j[0] = 0
&j[0] = 0x7ffdc879d640, j[0] = 0
```

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage Class of variables
- Function usage example
- Recursion

How to use functions: Example

- An Example
  - Goldbach's Conjecture
  - Any even number larger than 2 can be expressed as sum of two prim numbers
- It is not proved yet!
  - 1,000,000$ to proof ;-)
- Write a program that takes a set numbers which ends by 0 and checks correctness of the conjecture

Main Overall Algorithm

```
While(number is not zero)
        if(number >= 2 and even)
            Check Goldbach's Conjecture
        else
            Print some message
        read next number
```

This is a module

It is a black-box in this step

Check Goldbach's Conjecture Algorithm

Algorithm: Goldbach
Input: n
Output: 0 if conjecture is incorrect else 1

```
for(i from 2 to n/2)
    j = n – i
    if(is_prime(j))
        conjecture is correct

    i = next_prime_number(i)

Conjecture is incorrect
```

This is a module

It is a black-box in this step

is_prime algorithm

Algorithm: is_prime

Input: n

Output: 1 if n is prime else 0

```
for(i from 2 to sqrt(n))
   if(n % i == 0)
        n is not prime

n is prime
```

next_prime_number algorithm

Algorithm: next_prime_number
Input: n
Output: prime number

```
if n is 2
    output is 3
else
    do
        n = n + 2
    while(is_prime(n) == 0)
    output is n
```

Putting them altogether

```
int is_prime(int n){
  …
}

int next_prime_number(int n){
  …
}

int check_Goldbach(int n){
  …
}

int main(void){
  …
}
```

What We Will Learn

- Introduction
- Passing input parameters
- Producing output
- Scope of variables
- Storage Class of variables
- Function usage example
- Recursion

Introduction

- Iteration vs. Recursion
- Factorial
  - n! = n x n-1 x … x 2 x 1
  - n! = n x (n-1) !
- GCD
  - GCD(a, b) = Euclidean Algorithm
  - GCD(a, b) = GCD(b, a mod b)

Introduction

- Original problem can be solved by
  - Solving a similar but simpler problem (recursion)
    - (n-1)! in factorial, GCD(b, b mod a)
- There is a simple (basic) problem which we can solve it directly (without recursion)
  - Factorial: 1! = 1
  - GCD: b == 0

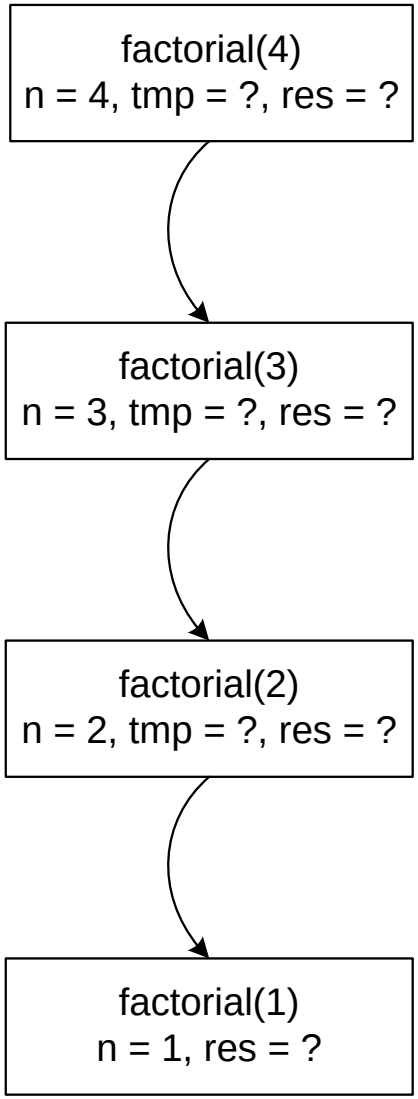
Recursion in C

- Recursive Algorithm
  - An algorithm uses itself to solve the problem
  - There is a basic problem with known solution
- Recursive Algorithms are implemented by recursive functions
- Recursive function
  - A function which calls itself
  - There is a condition that it does not call itself

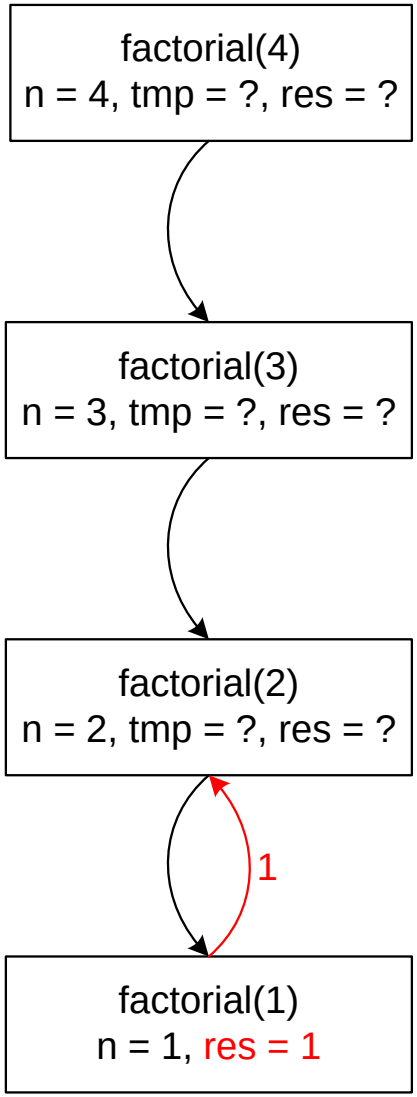
تابع بازگشتی برای محاسبه فاکتوریل

```
#include <stdio.h>

int factorial(int n){
   int res, tmp;
   if(n == 1)
       /* The basic problem */
       res = 1;
   else{
        /* recursive call */
       tmp = factorial(n - 1);
       res = n * tmp;
   }
   return res;
}

void main(void){
   int i = 4;
   int fac = factorial(i);
   printf("%d! = %d\n", i, fac);
}
```

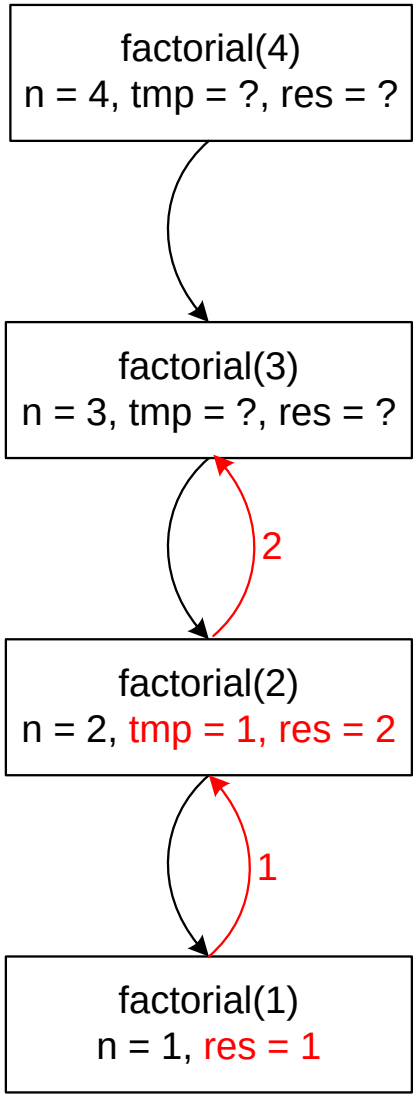
Function Call Graph + Stacks



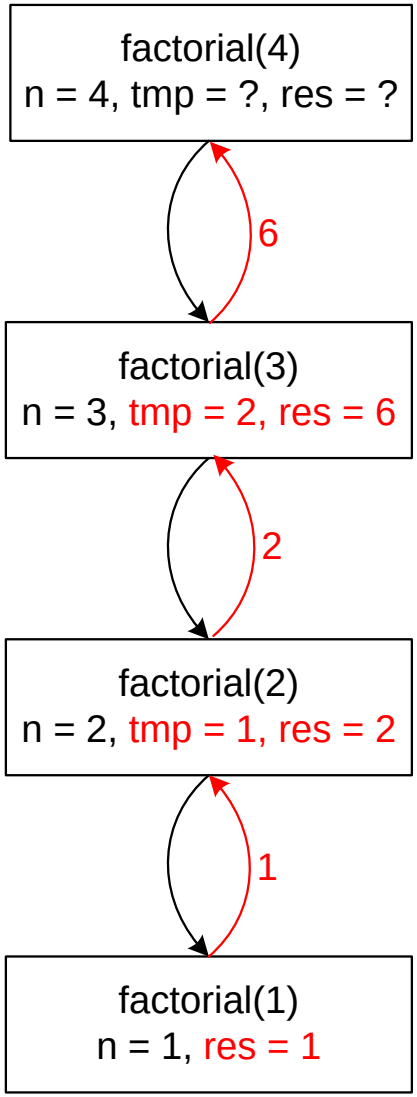
Function Call Graph + Stacks



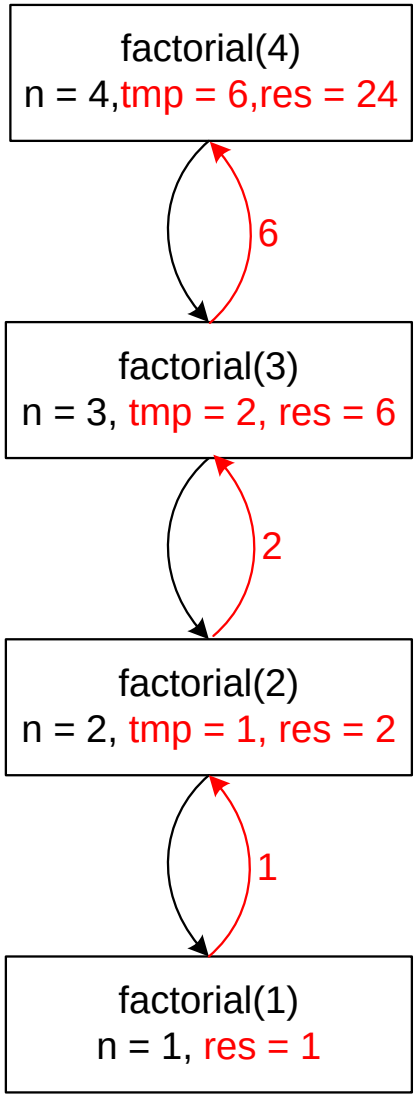
Function Call Graph + Stacks



Function Call Graph + Stacks



Function Call Graph + Stacks



Examples

- Recursive version of GCD?
- Recursive version of Fibonacci numbers
  - Fibonacci numbers
    - 1, 1, 2, 3, 5, 8, ...
- Print digits: left-to-right and right-to-left

تابع بازگشتی محاسبه ب.م.م

```
#include <stdio.h>

int GCD(int a, int b){
    if(b == 0)
        return a;
    else
        return GCD(b, a % b);
}

int main(void){

    printf("GCD(1, 10) = %d \n", GCD(1, 10));
    printf("GCD(10, 1) = %d \n", GCD(10, 1));
    printf("GCD(15, 100) = %d \n", GCD(15, 100));
    printf("GCD(201, 27) = %d \n", GCD(201, 27));

    return 0;
}
```

تابع بازگشتی محاسبه
جمله-nام اعداد فیبوناچی

```
#include <stdio.h>

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

int main(void){
    printf("fibo(1) = %d\n", fibo(1));
    printf("fibo(3) = %d\n", fibo(3));
    printf("fibo(5) = %d\n", fibo(5));
    printf("fibo(8) = %d\n", fibo(8));

    return 0;
}
```

تابع بازگشتی چاپ ارقام از راست به چپ

```
#include <stdio.h>

void print_digit_right_left(int n){
    int digit = n % 10;
    printf("%d  \n", digit);
    if(n >= 10)
        print_digit_right_left(n / 10);
}

int main(void){

    printf("\n print_digit_right_left(123): ");
    print_digit_right_left(123);

    printf("\n print_digit_right_left(1000): ");
    print_digit_right_left (1000);

    return 0;
}
```

تابع بازگشتی چاپ ارقام از چپ به راست

```
#include <stdio.h>

void print_digit_left_right(int n){
    if(n >= 10)
         print_digit_left_right(n / 10);

    int digit = n % 10;
    printf("%d  \n", digit);
}

int main(void){

    printf("\n print_digit_left_right(123): ");
    print_digit_left_right(123);

    printf("\n print_digit_left_right(1000): ");
    print_digit_left_right (1000);
   return 0;
}
```

Indirect recursion

- What we have seen are direct recursion
  - A function calls itself directly
- Indirect recursion
  - A function calls itself using another function
  - Example:
    - Function A calls function B
    - Function B calls function A

تابع بازگشتی تعیین زوج یا فرد بودن عدد

```
#include <stdio.h>
#include <stdbool.h>
bool is_even(int n);
bool is_odd(int n);

bool is_even(int n){
    if(n == 0)
        return true;
    if(n == 1)
        return false;
    else
        return is_odd(n - 1);
}
bool is_odd(int n){
     if(n == 0)
         return false;
     if(n == 1)
         return true;
     else
        return is_even(n - 1);
}
```

```
int main(void){

    if(is_even(20))
        printf("20 is even\n");
    else
        printf("20 is odd\n");

   printf("23 is %s\n", is_odd(23) ? "odd" : "even");

   return 0;
}
```

Bugs & Avoiding Them

- Be careful about the order of input parameters

```
int diff(int a, int b){return a - b;}
```

```
diff(x,y) or diff(y,x)
```

- Be careful about casting in functions
- Recursion must finish, be careful a bout basic problem in the recursive functions
  - No base problem → Stack Overflow
- Static variables are useful debugging

Reference

- Reading Assignment: Chapter 5 of “C How to Program”

Homework

- HW 5