

計算機程式設計

Computer Programming

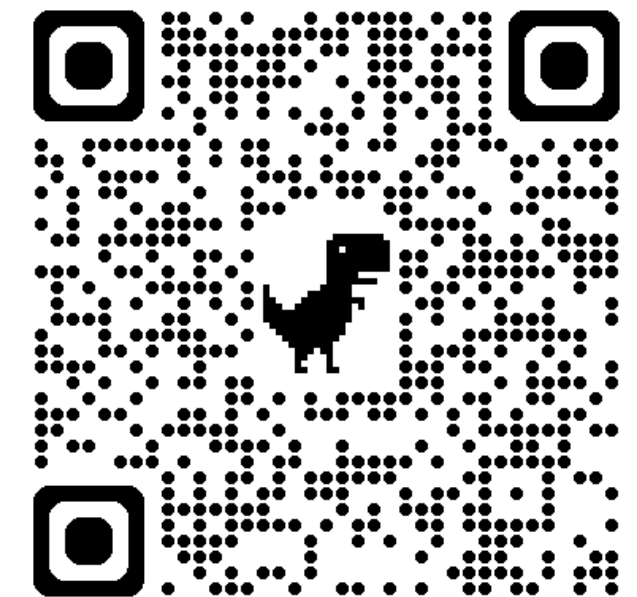
Functions

Instructor: 林英嘉

2024/10/28



[W8 Slido: #9798754](#)



[GitHub repo](#)

Outline

- Review
 - Usage of “return”
- Internal properties when passing an array to a function
- Recursion
- Function scope
-

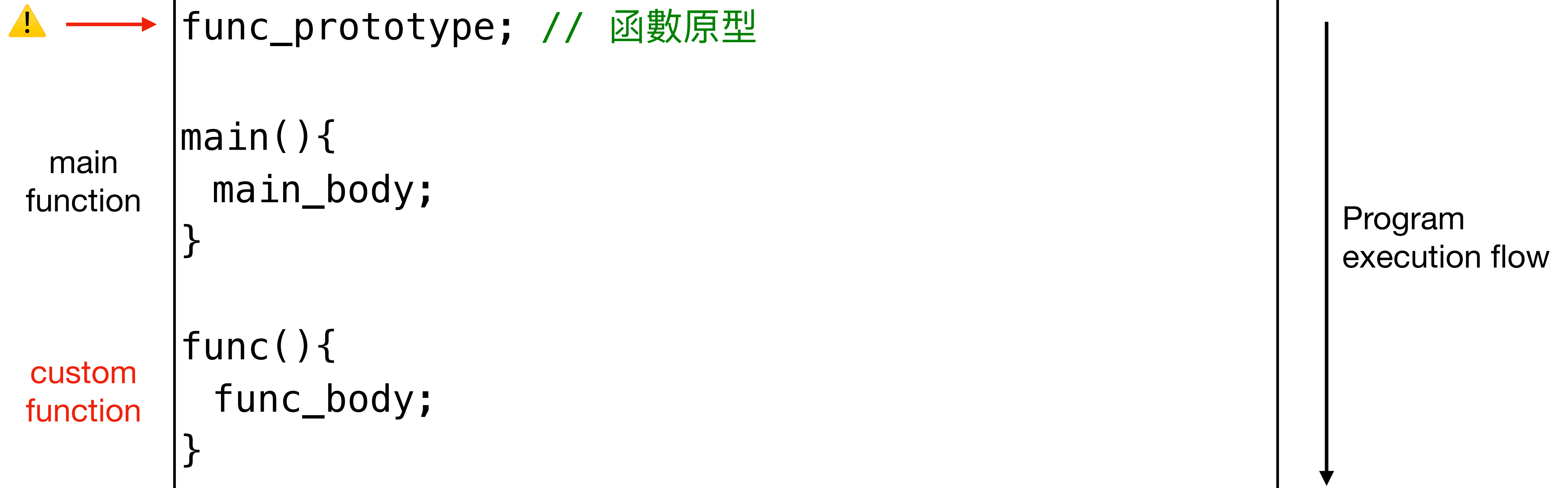
Categories

- [Definition]: 定義 (通常一堆字)
- [Declaration]: 宣告方法 (通常 pseudo code)
- [Usage]: 使用方法 (通常 pseudo code)
- [Illustration]: 我有畫圖
- [Important Notes]: 注意事項 (通常一堆字)
- [Notes]: 注意事項 (通常一堆字)，但重要程度較 Important Notes 低
- 什麼都沒標: 真的 code 且右上角會有灰字告訴你該 code 在 GitHub repo 的位置

Review (Functions)

[Declaration] How to write a function

You can declare a function with a **function prototype**



- Then, your program will first check the function prototype and use the function you define outside the main function.

[Declaration] Function Prototype (函數原型)

- A function prototype is:

```
return_type func_name(type1, type2, ...);
```

- Purposes:

1. **Type Checking:** Help the compiler **check the correctness of data types** when you use a function in the main function.
2. **Function Declaration:** Allows function calls before the function is defined.

- You can also write a function prototype as the following to increase readability:

```
return_type func_name(type1 param1, type2 param2, ...);
```

[Declaration] Input Array to a Function

prototype

```
return_type func_name(type arr[], type2 param2, ...)
```

main
function

```
int main(void){  
    array_declaration;  
    ...  
}
```

↑
Not
required

custom
function

```
return_type func_name(type arr[], type2 param2, ...){  
    body;  
    return value;  
}
```

↑
Not
required

Addresses between Parameters and Arguments

C-course-materials/05-Functions/array.c

```
#include <stdio.h>

void modifyArray(int arr[]) {
    printf("Address of parameter 'arr' inside function: %p\n", arr);
    arr[0] = 999; // Modify the first element
}

int main(void) {
    int score[4] = {80, 85, 90, 100};
    printf("Address of argument 'score' in main: %p\n", score);

    printf("First element before modification: %d\n", score[0]);
    modifyArray(score);
    printf("First element after modification: %d\n", score[0]);
    return 0;
}
```


Print the memory address of an array

C-course-materials/04-Arrays/print_array_addr.c

```
#include <stdio.h>
int main(){
    int score[4] = {80, 85, 90, 100};
    printf("The address of this array: %p\n", &score);
    for (int i = 0; i < 4; i++){
        printf("The address of score[%d]: %p\n", i, &score[i]);
    }
}
```

Output

```
The address of this array: 0x7fffffffdd80
The address of score[0]: 0x7fffffffdd80
The address of score[1]: 0x7fffffffdd84
The address of score[2]: 0x7fffffffdd88
The address of score[3]: 0x7fffffffdd8c
```

Usage of “return”

[Definition] What is “return”?

- return (回傳) usually comes with a statement, which is called a **return statement**.
- A return statement ends the execution of a function, and returns control to the calling function.

This line is a
return statement.

```
return_type func_name(type1 param1, type2 param2, ...){  
    body;  
    return value;  
}
```

[Illustration] Usage of “return”

```
#include <stdio.h>
int add(int a, int b){
    return a + b;
}
int main(void){
    int sum;
    sum = add(5, 6);
    printf("Sum: %d", sum);
    return 0;
}
```

Declare `add` with a return value



Call `add` in the main function



Obtain the returned value



Other things ...

[Notes / Recap] Property of an Array

- The return type of a function is the type of value that the function returns.
- Omitting the return type is **not recommended**. Placing a return type increases readability and lowers the chance of producing wrong behaviors.

[Notes / Recap] Property of an Array

C-course-materials/05-Functions/return_example.c

- The return type of a function is the type of value that the function returns.
- Omitting the return type is not recommended. Placing a return type increases readability and lowers the chance of producing wrong behaviors.
- Yes, **you can omit the return type**. But a function will set the return type as **integer** in default.
- If you don't need to return a value (i.e. just print), omitting the return type (void) does not cause an undesired result.

**Internal properties when passing
an array to a function**

將陣列輸入到函數時的內部行為

[Usage] Calling a Function with an array

```
void modifyArray(int arr[]) {  
    arr[0] = 999; // Modify the first element  
}
```

✓ This is correct for calling `modifyArray` in the main function.

```
int main(void) {  
    int score[4] = {80, 85, 90, 100};  
    modifyArray(score);  
}
```

✗ This is not correct for calling `modifyArray` in the main function.

```
int main(void) {  
    int score[4] = {80, 85, 90, 100};  
    modifyArray(score[0]);  
}
```


[Illustration] 1D Array as a Argument

```
//Input score  
modifyArray(score);
```

- Input an array as an argument to a function is actually **passing its memory address**.

0x7fffffffdd80

score[0]	score[1]	score[2]	score[3]
----------	----------	----------	----------

0x7fffffffdd80

0x7fffffffdd84

0x7fffffffdd88


0x7fffffffdd8c

```
void modifyArray(int arr[])
```

Sub-Array as a Argument

C-course-materials/05-Functions/func_pass_array.c

```
#include <stdio.h>
void print_array(int arr[]) {
    for (int i = 0; i < 4; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nFinish printing an array\n");
}
void modifyArray(int arr[]) {
    printf("Address of parameter 'arr' inside function: %p\n", arr);
    arr[0] = 999; // Modify the first element
}
int main(void) {
    int score[4] = {80, 85, 90, 100};
    print_array(score);
    modifyArray(score);
    print_array(score);
    modifyArray(&score[1]);
    print_array(score);
}
```



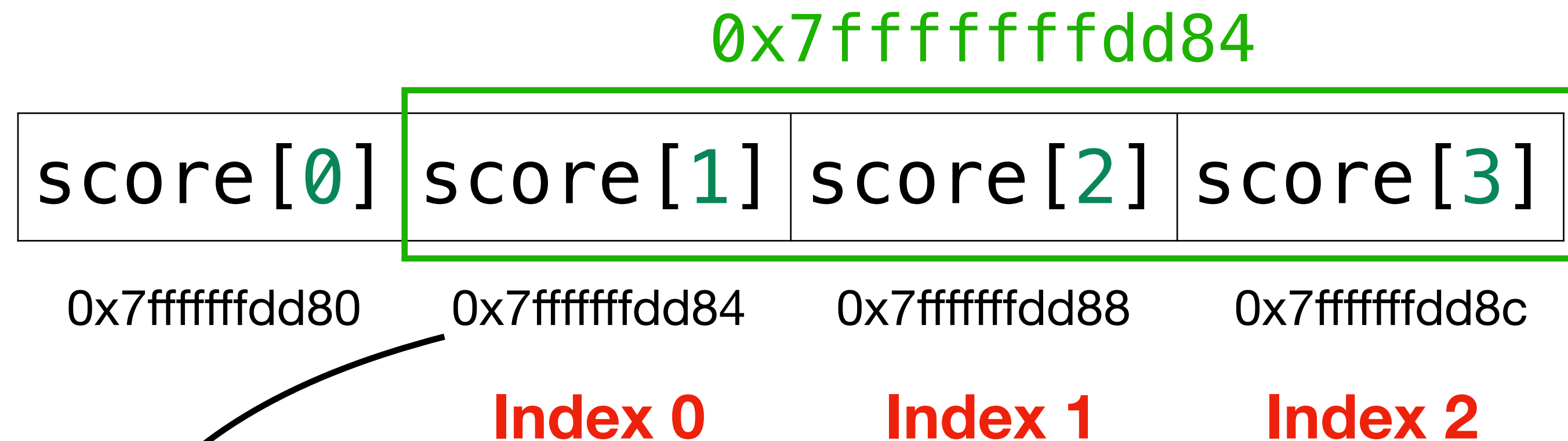
[Important Notes / Recap] Property of an Array

- An array **itself** is **a memory address**. (陣列名稱本身就是存放陣列位址的變數)
 - Example: `printf("%p", score);` will show the address of ``score``.
- **Array subscripting** gets the value of an element in an array.
 - Example: `score[0]` will get a value of 80.
- Using the address operator (`&`) returns with array subscripting gets the **address of an element** in an array.
 - Example: `printf("%p", &score[0]);` will show the address of ``score[0]``.

[Illustration] 2D Array as a Argument

```
//Input a sub-array  
modifyArray(&score[1]);
```

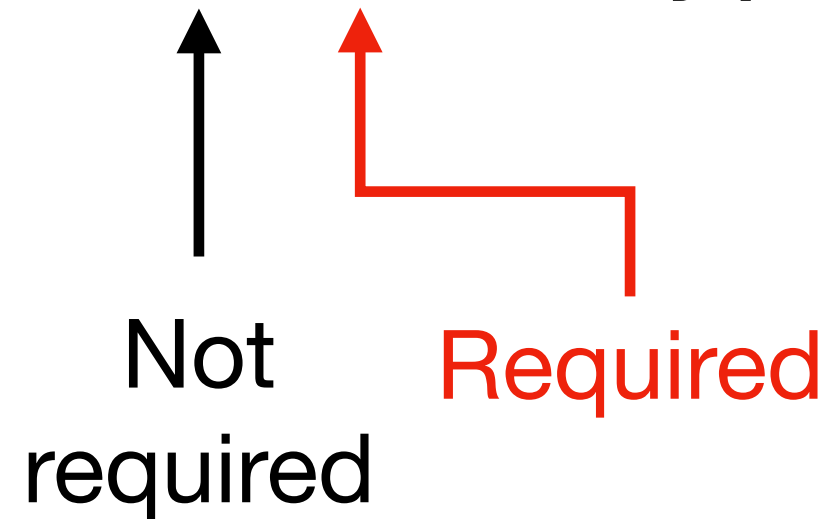
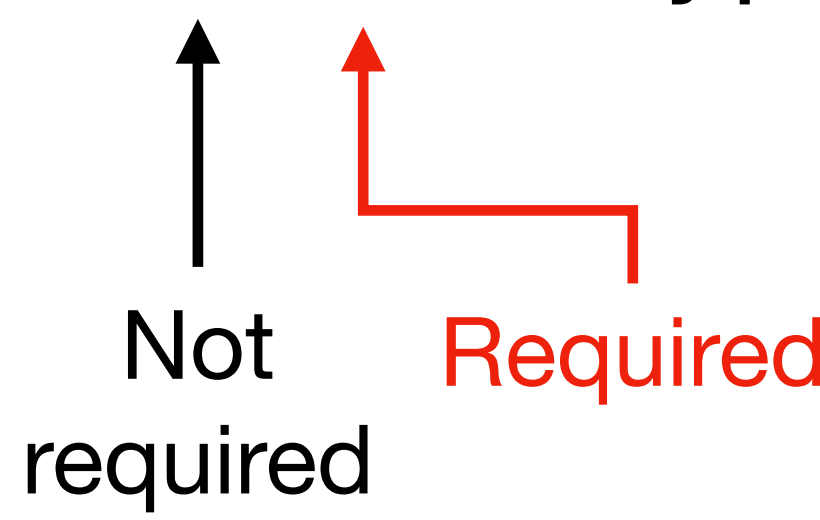
- Input an array as an argument to a function is actually passing its memory address.



```
void modifyArray(int arr[])
```

[Declaration] Input a 2D Array to a Function

- Example: pass `score[2][4]` to a function You can only leave **the leftmost index empty!**

prototype	<code>return_type func_name(type arr[][4], type2 param2, ...)</code>	
		
		Not required Required
main function	<pre>int main(void){ array_declaration; ... }</pre>	
custom function	<pre>return_type func_name(type arr[][4], type2 param2, ...){ body; return value; }</pre>	
		Not required Required

Passing a 2D array to a function

C-course-materials/05-Functions/func_pass_2Darray.c

```
#include <stdio.h>
void print_array(int arr[][4]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    printf("Finish printing an array\n");
}
void modifyArray(int arr[][4]) {
    printf("Address of parameter 'arr' inside function: %p\n", arr);
    arr[0][3] = 999; // Modify the first element
}
int main(void) {
    int score[][4] = {
        {80, 85, 90, 100},
        {60, 65, 70, 100}
    };
    print_array(score);
    modifyArray(score);
    print_array(score);
}
```

[Declaration] Pass Array to a Function with length

- For a 1D array:

custom
function

```
return_type func_name(type arr[], int n){  
    for (int i = 0; i < n; i++){  
        ...  
    }  
}
```

- For a 2D array (Example: `score[2][4]`):


custom
function

```
return_type func_name(type arr[][4], int num_rows){  
    for (int i = 0; i < num_rows; i++){  
        for (int j = 0; j < 4; j++){  
            ...  
        }  
    }  
}
```

Pass an array with a length parameter

C-course-materials/05-Functions/func_pass_array_with_length.c

Length parameter



```
#include <stdio.h>
void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nFinish printing an array\n");
}
void modifyArray(int arr[]) {
    printf("Address of parameter 'arr' inside function: %p\n", arr);
    arr[0] = 999; // Modify the first element
}
int main(void) {
    int score[4] = {80, 85, 90, 100};
    print_array(score, 4);
    modifyArray(score);
    print_array(score, 4);
}
```


Recursion (逕迴)

[Definition] What is recursion?

- We are now talking about “**recursion** in a function”.
- Recursion is to call the same function inside a function. (自己呼叫自己)

```
int do_func(int n){  
    ...;  
    do_func(n);  
}  
  
int main(void){  
    int var_a;  
    do_func(var_a);  
}
```

[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

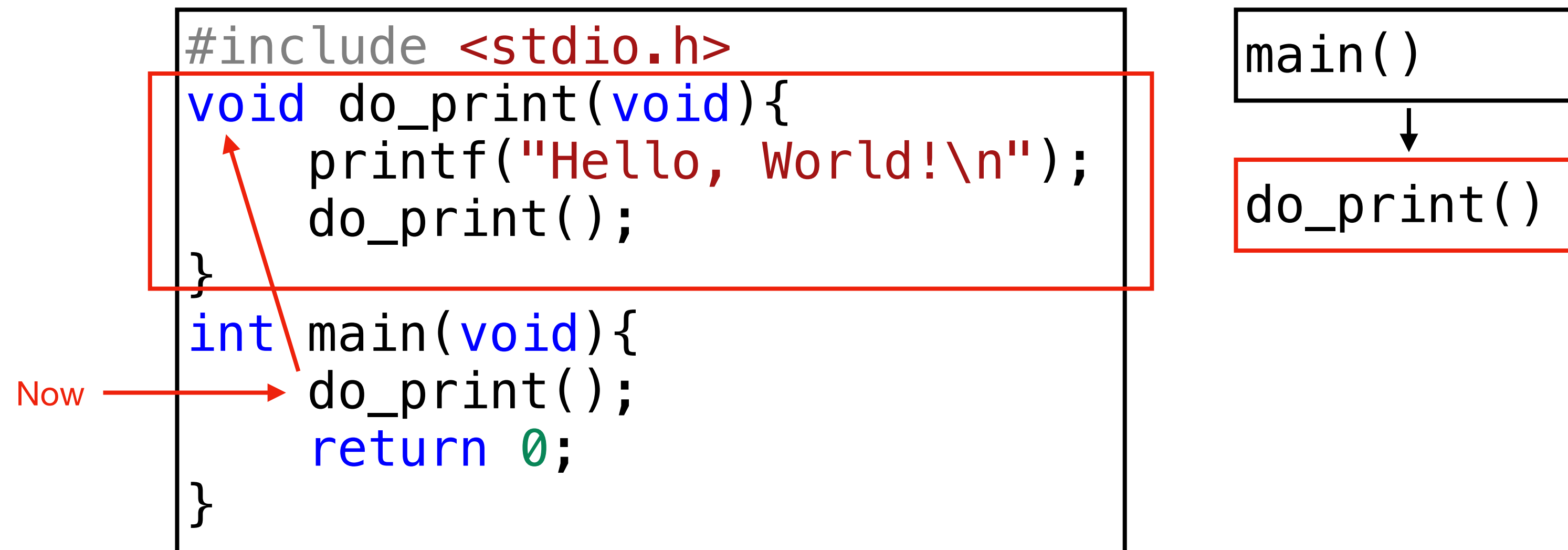
Start →

```
#include <stdio.h>
void do_print(void){
    printf("Hello, World!\n");
    do_print();
}
int main(void){
    do_print();
    return 0;
}
```

main()

[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

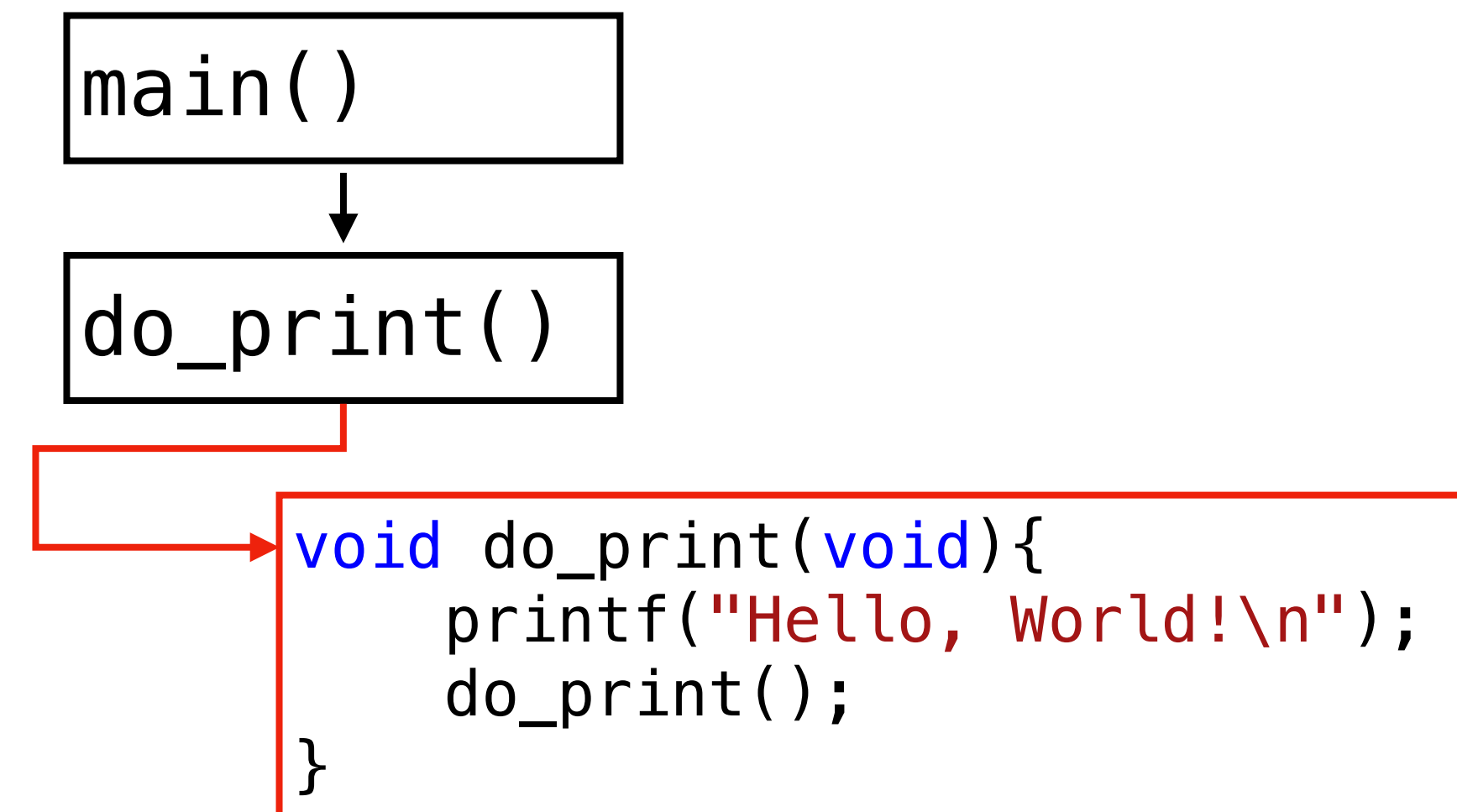


[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

Now →

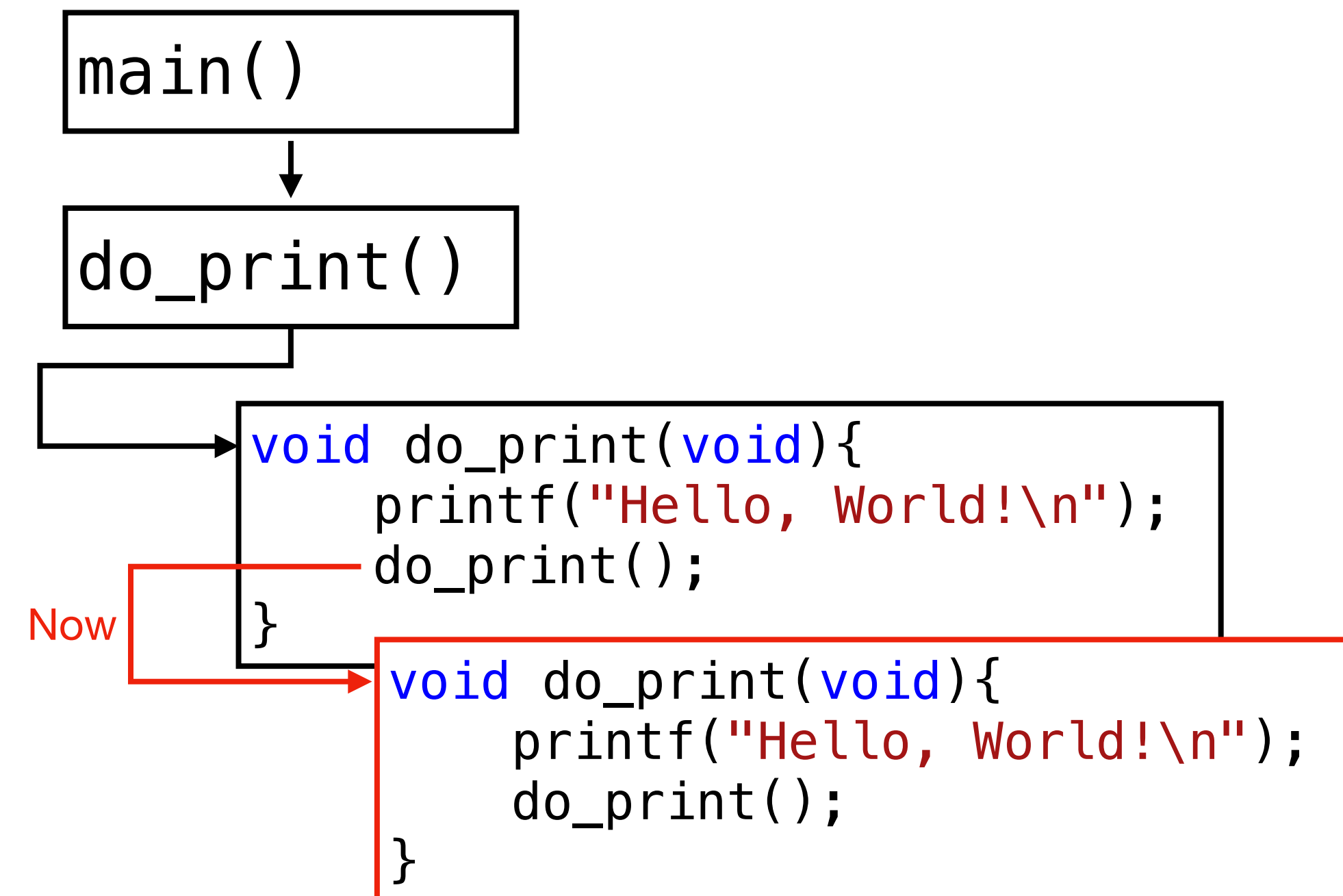
```
#include <stdio.h>
void do_print(void){
    printf("Hello, World!\n");
    do_print();
}
int main(void){
    do_print();
    return 0;
}
```



[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

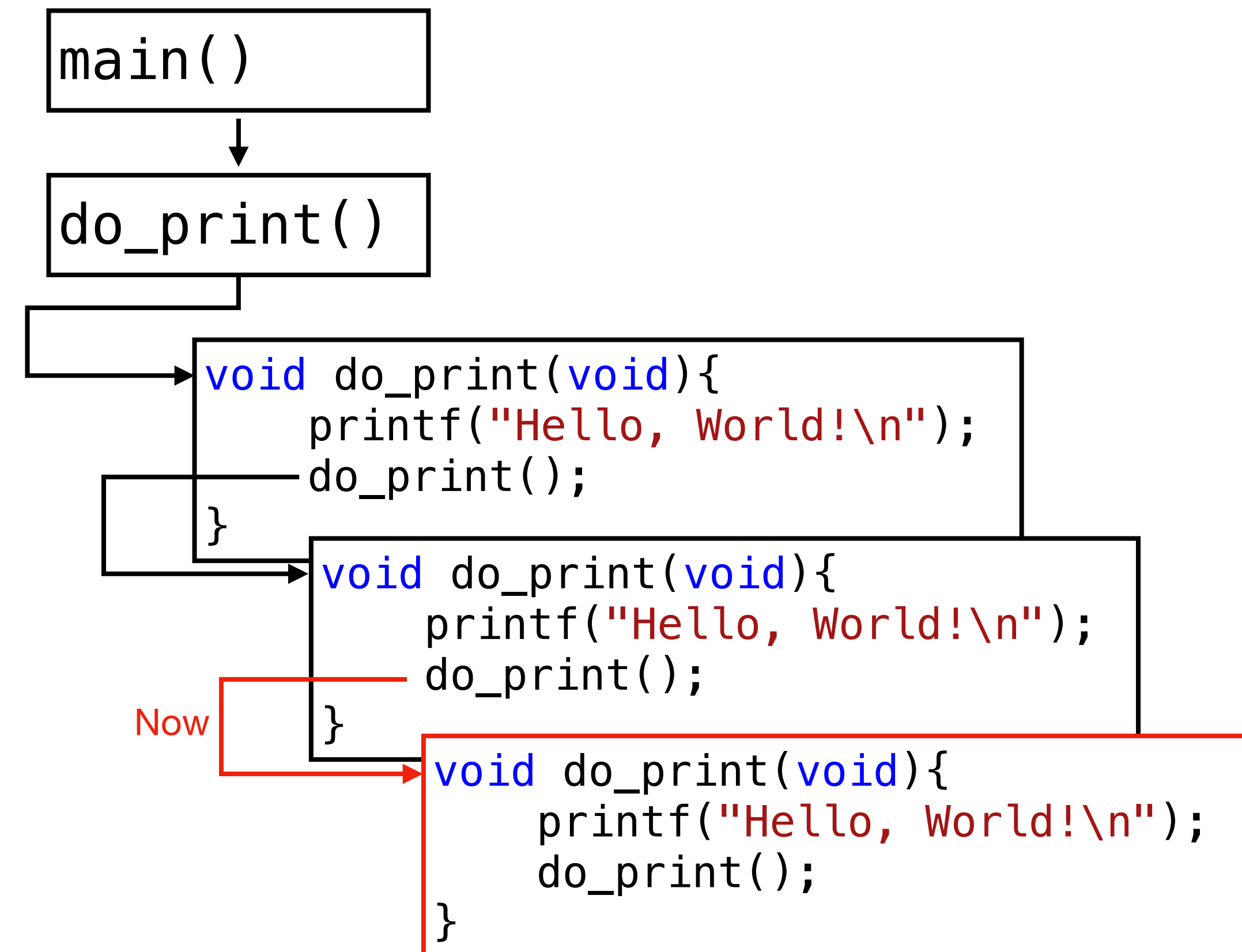
```
#include <stdio.h>
void do_print(void){
    printf("Hello, World!\n");
    do_print();
}
int main(void){
    do_print();
    return 0;
}
```



[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

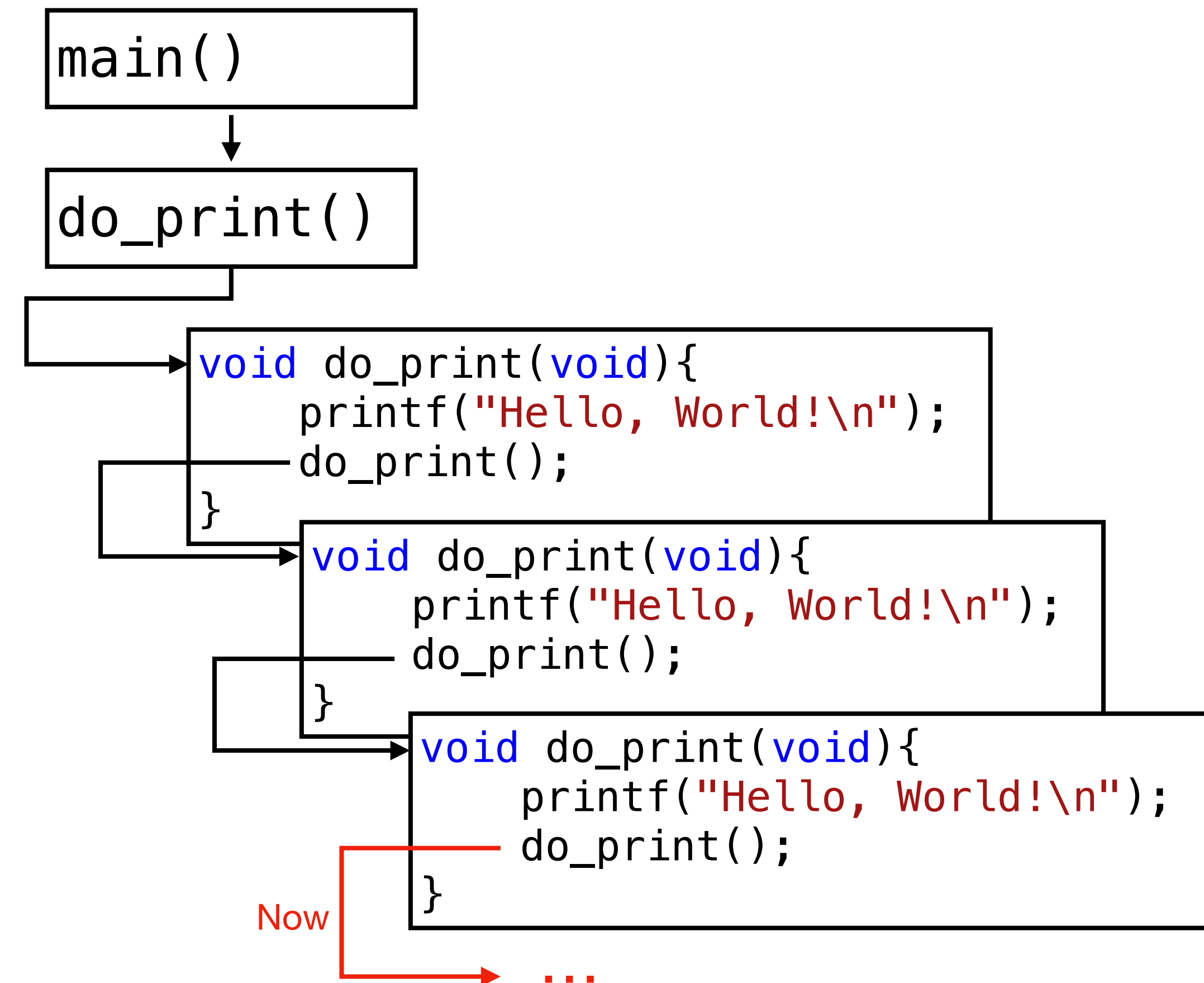
```
#include <stdio.h>
void do_print(void){
    printf("Hello, World!\n");
    do_print();
}
int main(void){
    do_print();
    return 0;
}
```



[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c

```
#include <stdio.h>
void do_print(void){
    printf("Hello, World!\n");
    do_print();
}
int main(void){
    do_print();
    return 0;
}
```



This program will not end!

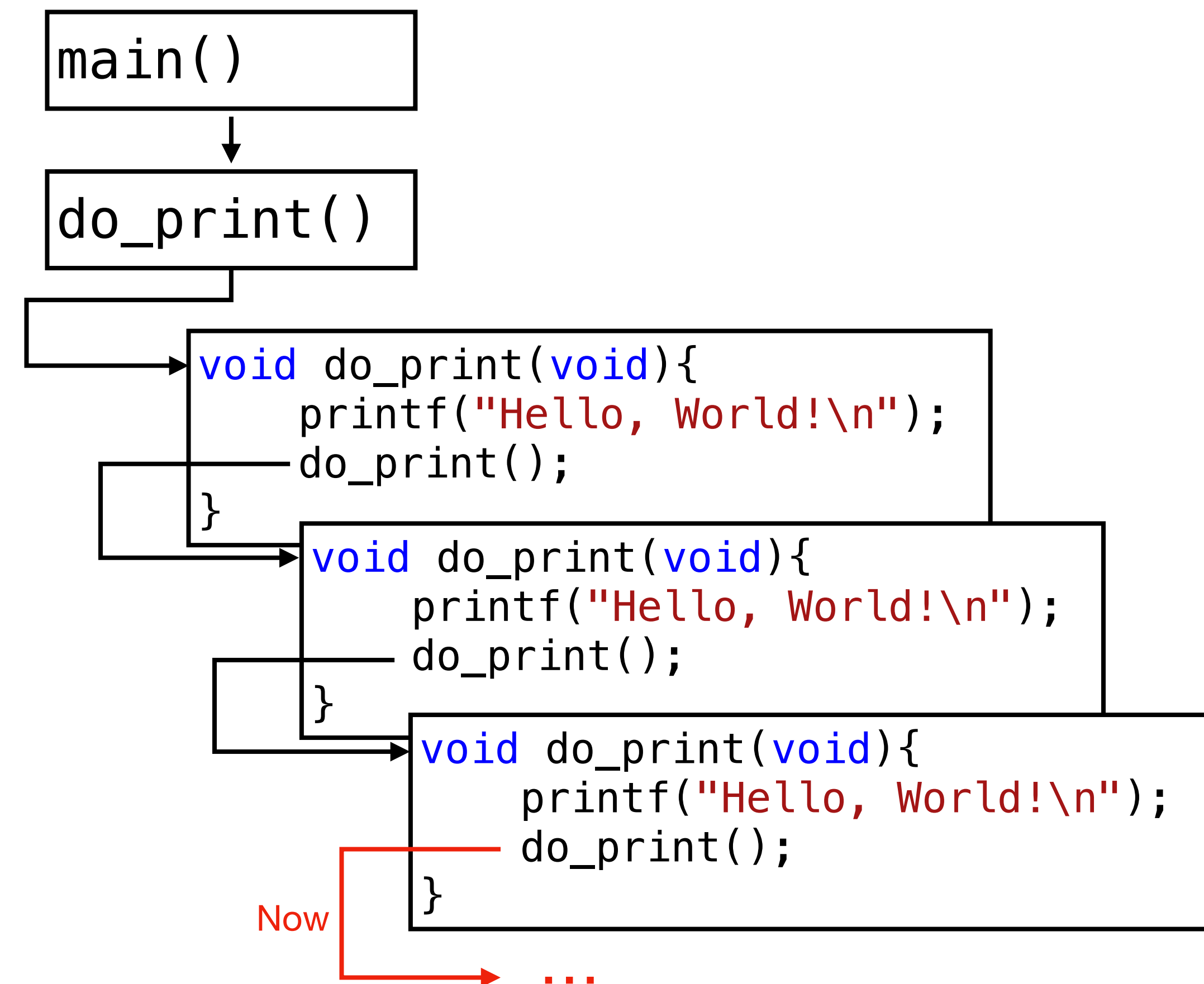
[Illustration] Recursion

C-course-materials/05-Functions/infinite_recursion.c



Figure source: <https://x.com/ProductHunt/status/1013695862508097536>

This program will not end!



[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```

[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

Start →

```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```

main()

[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```

Now

main()



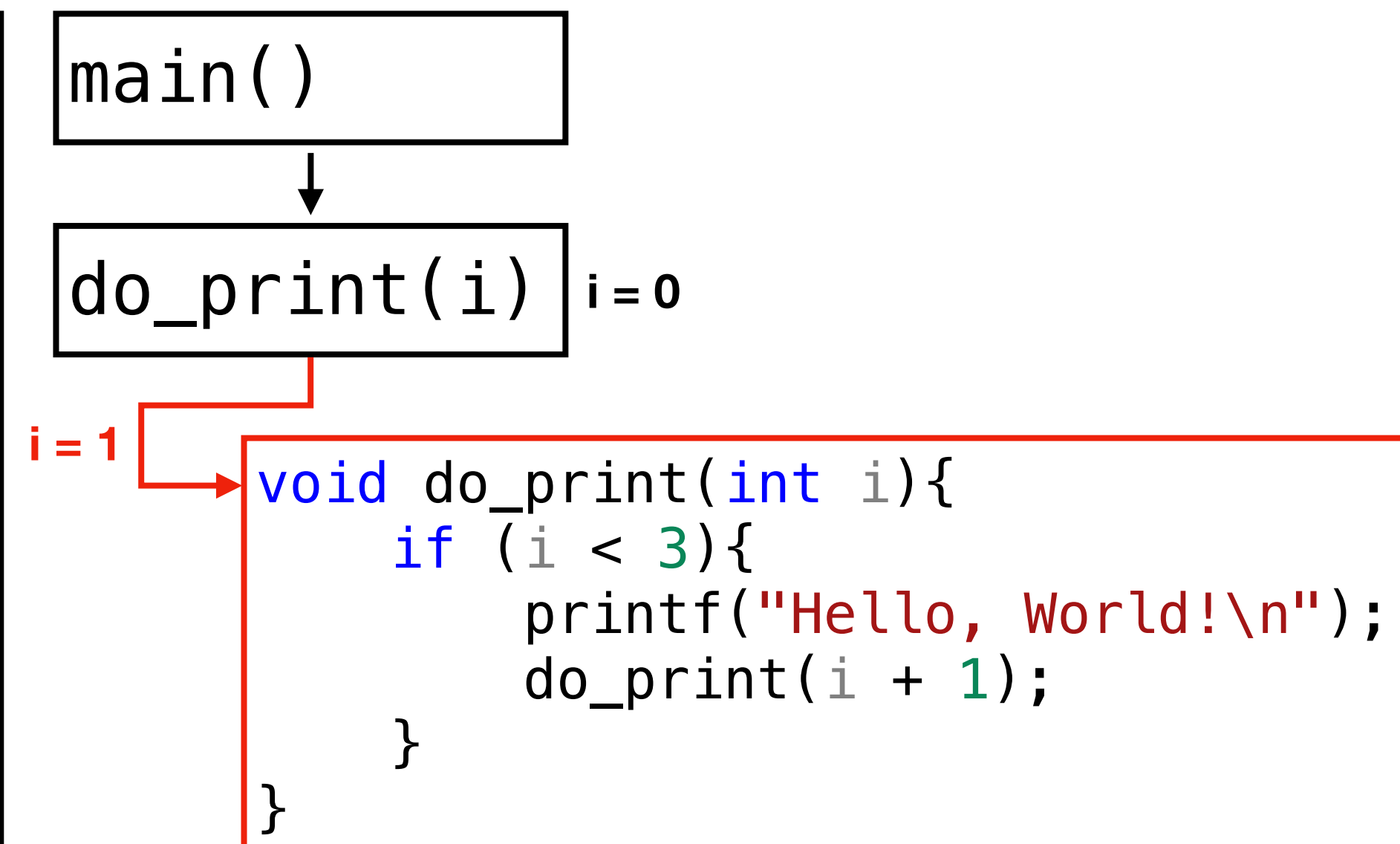
do_print(i) i=0

[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

Now →

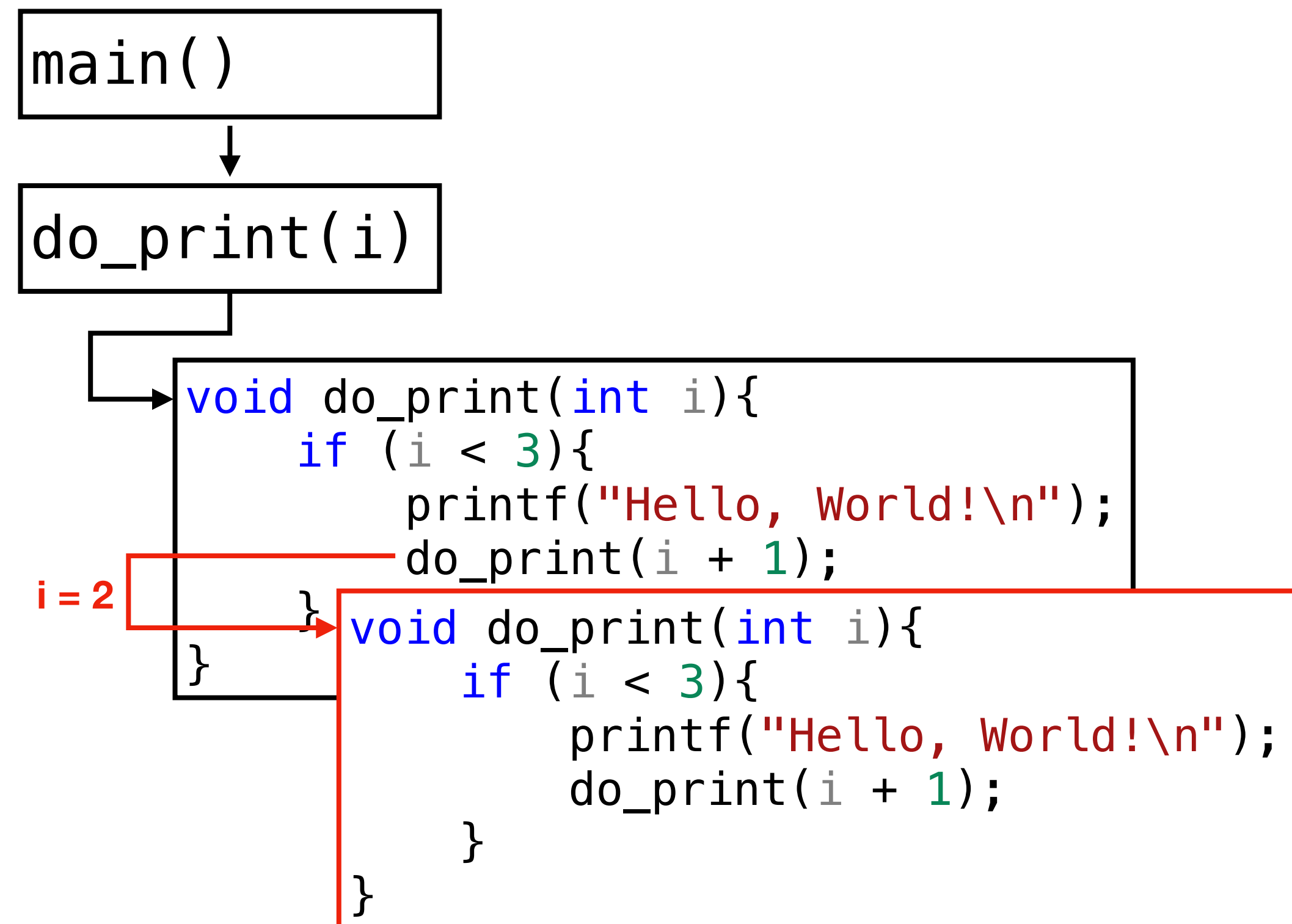
```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```



[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

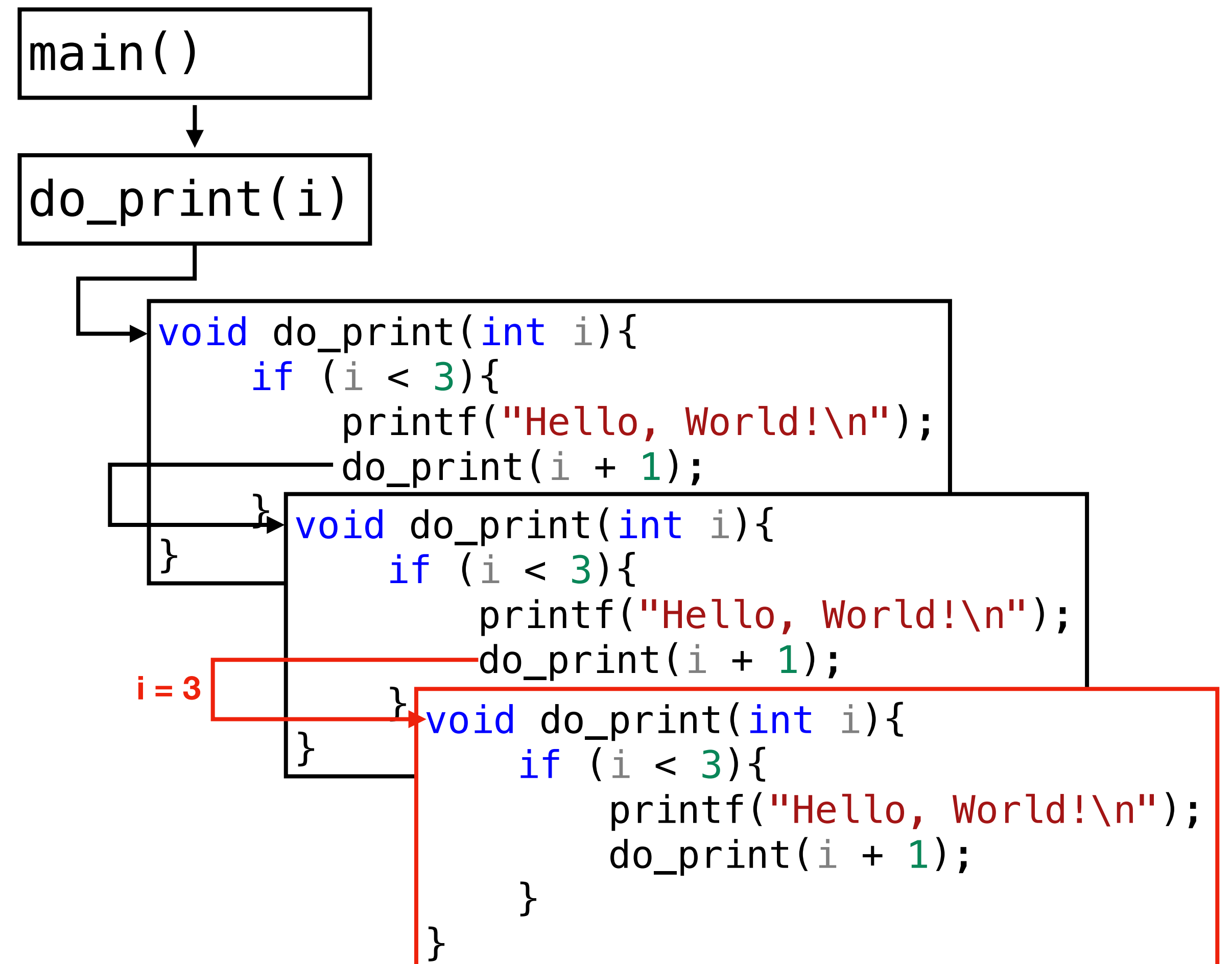
```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```



[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

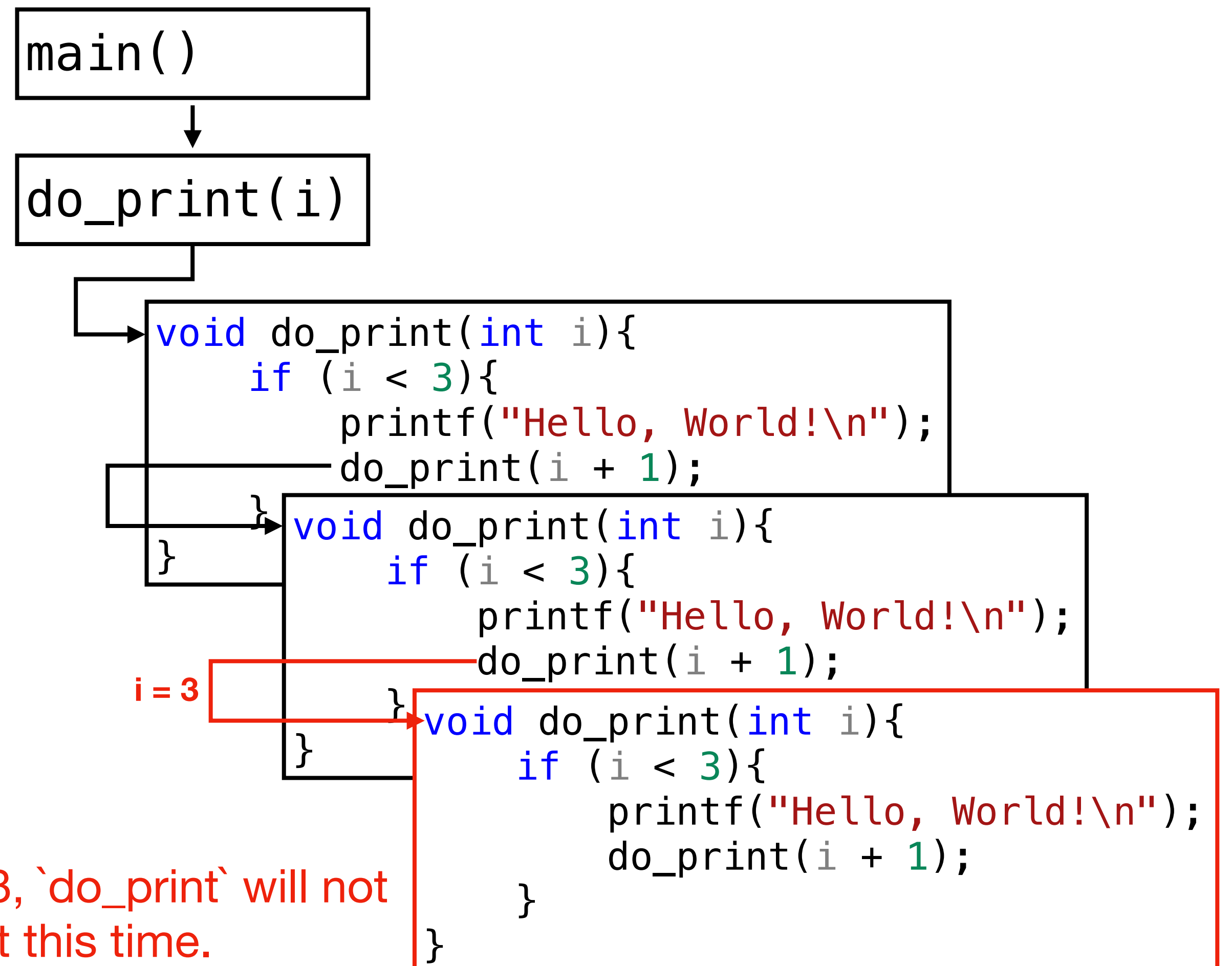
```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```



[Illustration] Recursion (finite version)

C-course-materials/05-Functions/finite_recursion.c

```
#include <stdio.h>
void do_print(int i){
    if (i < 3){
        printf("Hello, World!\n");
        do_print(i + 1);
    }
}
int main(void){
    int i = 0;
    do_print(i);
    return 0;
}
```



Because `i == 3`, `do_print` will not be executed at this time.

Practical Examples of Recursion

[Question] Factorial

Write a program that takes a positive integer n and outputs its factorial:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$$

- Input

- Output

[Illustration] Factorial

main()

Start →

```
#include <stdio.h>
int do_factorial(int n){
    if (n == 0){
        return 1;
    }
    return n * do_factorial(n - 1);
}
int main(void){
    int num = 3;
    int ans = do_factorial(num);
    printf("%d! = %d", num, ans);
    return 0;
}
```

[Illustration] Factorial

C-course-materials/05-Functions/factorial.c

```
#include <stdio.h>
int do_factorial(int n){
    if (n == 0){
        return 1;
    }
    return n * do_factorial(n - 1);
}
int main(void){
    int num = 3;
    int ans = do_factorial(num);
    printf("%d! = %d", num, ans);
    return 0;
}
```

Now →

main()



do_factorial(3)

[Illustration] Factorial

C-course-materials/05-Functions/factorial.c

```
#include <stdio.h>
int do_factorial(int n){
    if (n == 0){
        return 1;
    }
    return n * do_factorial(n - 1);
}
int main(void){
    int num = 3;
    int ans = do_factorial(num);
    printf("%d! = %d", num, ans);
    return 0;
}
```

Now

main()

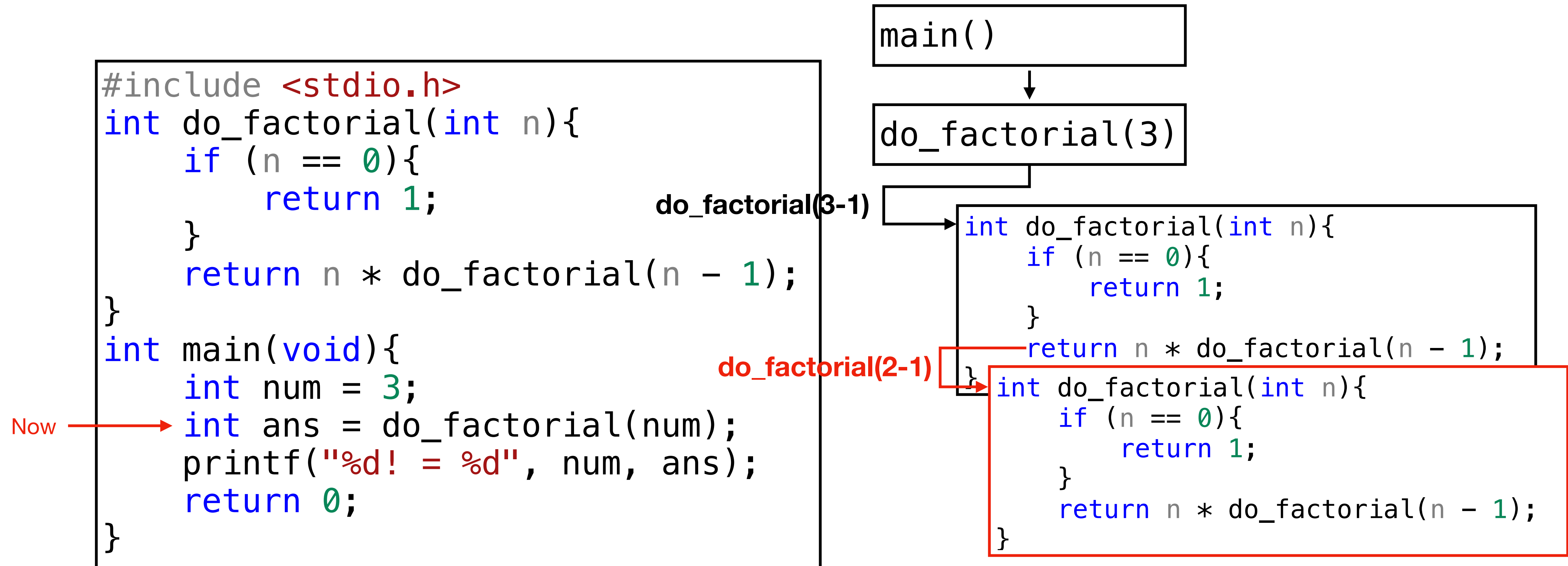
do_factorial(3)

do_factorial(3-1)

```
int do_factorial(int n){
    if (n == 0){
        return 1;
    }
    return n * do_factorial(n - 1);
}
```

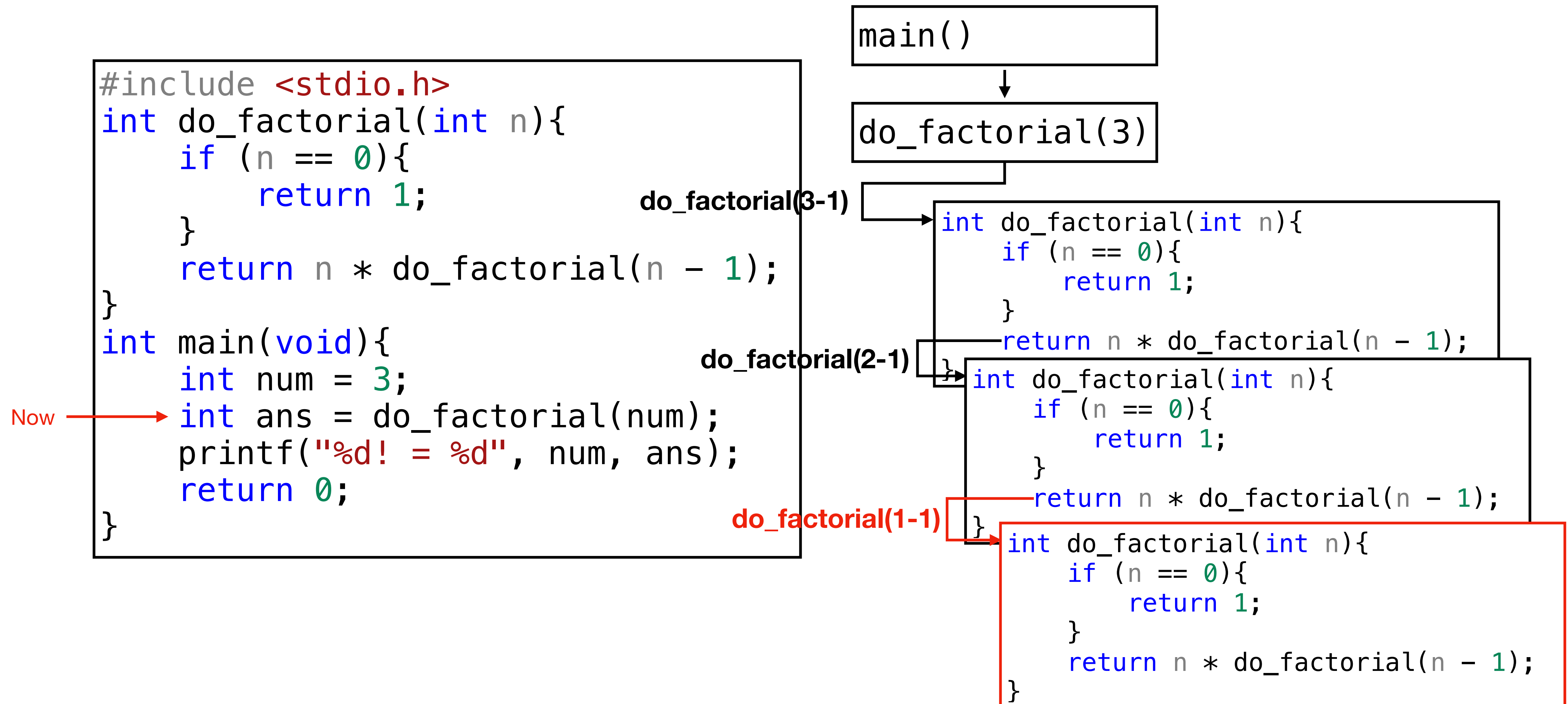
[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



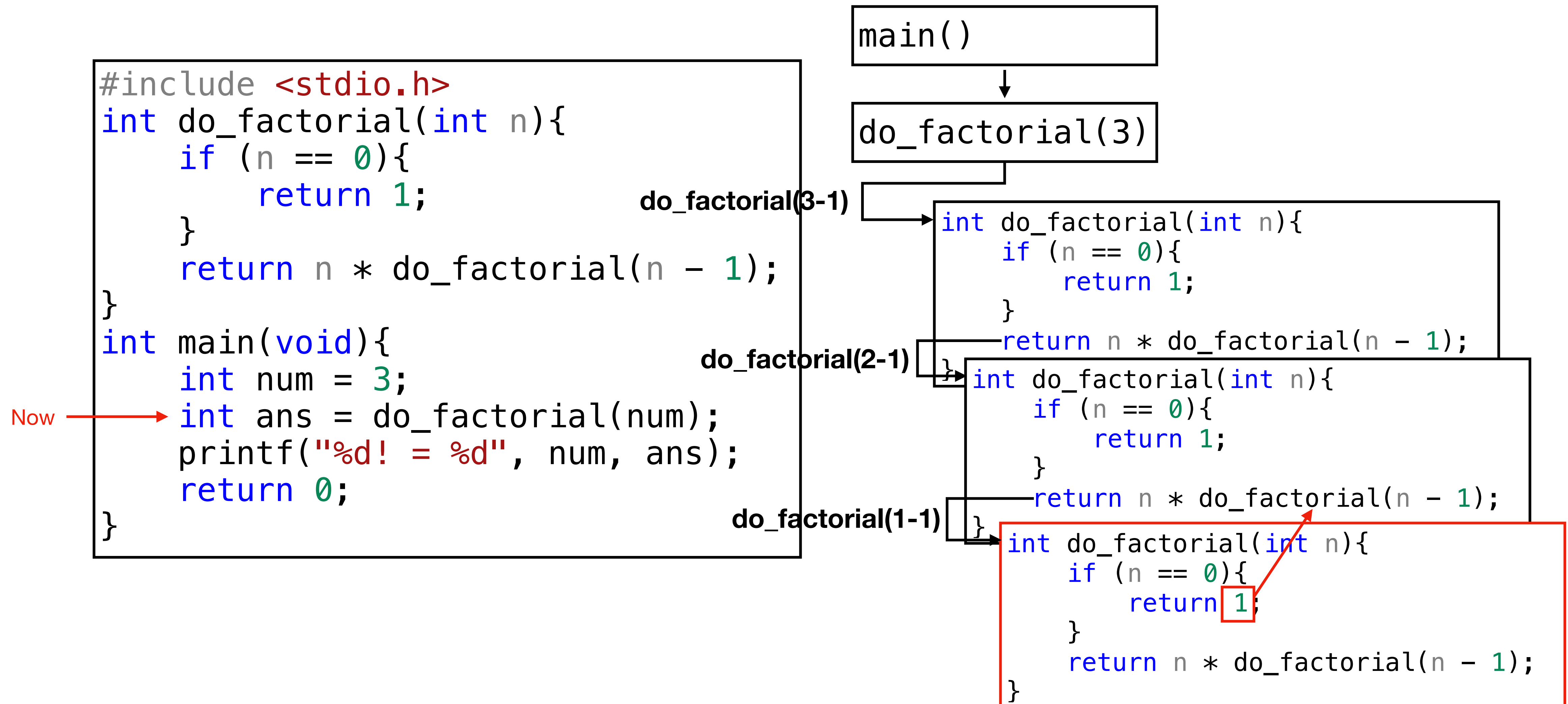
[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



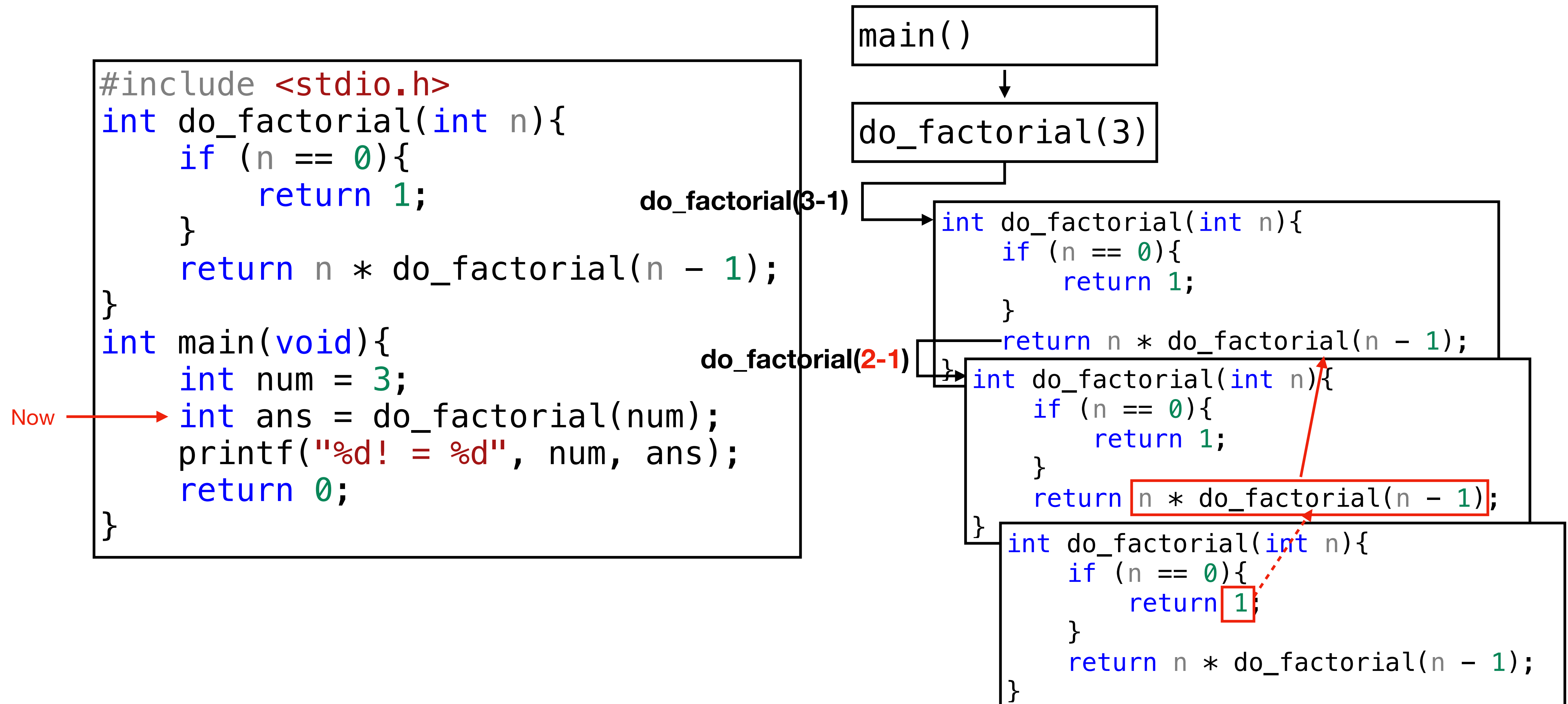
[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



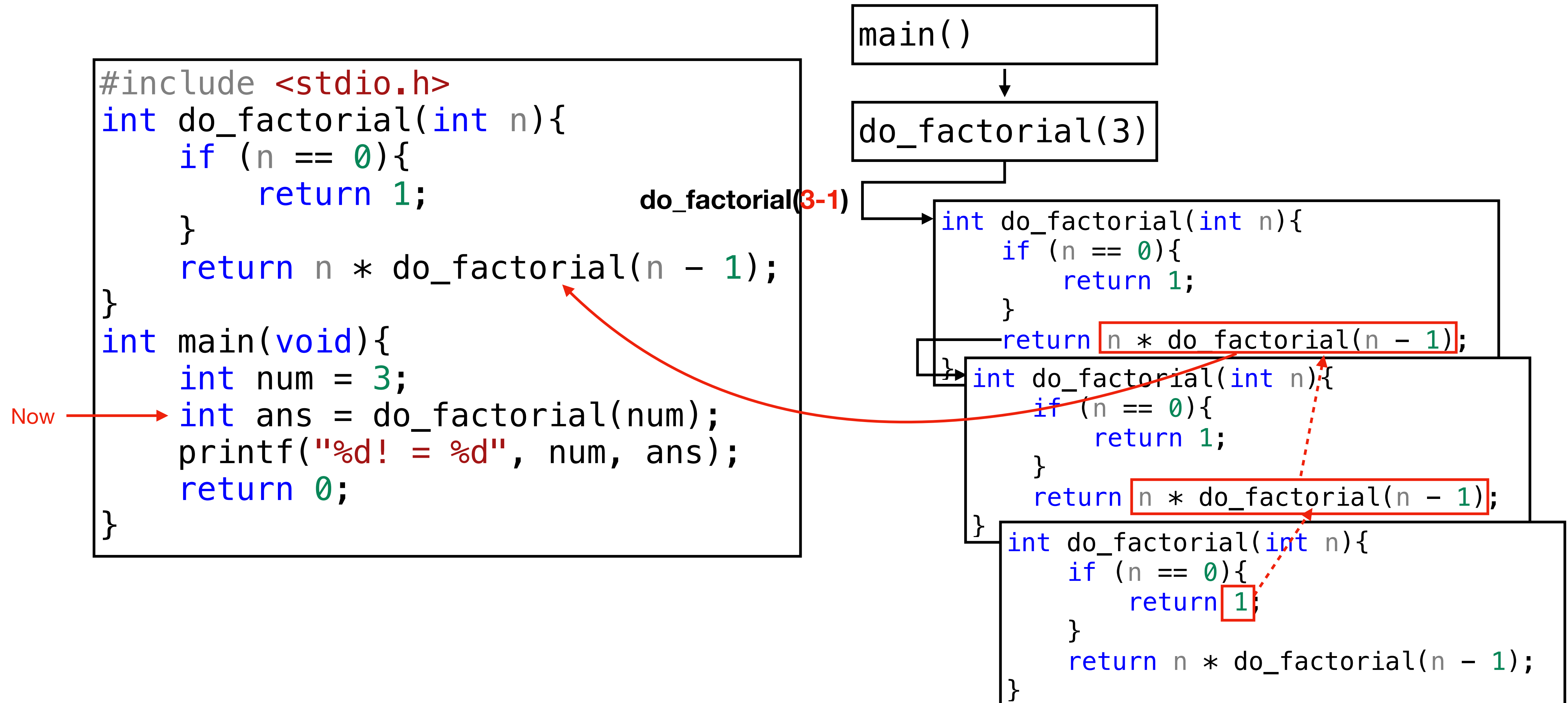
[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



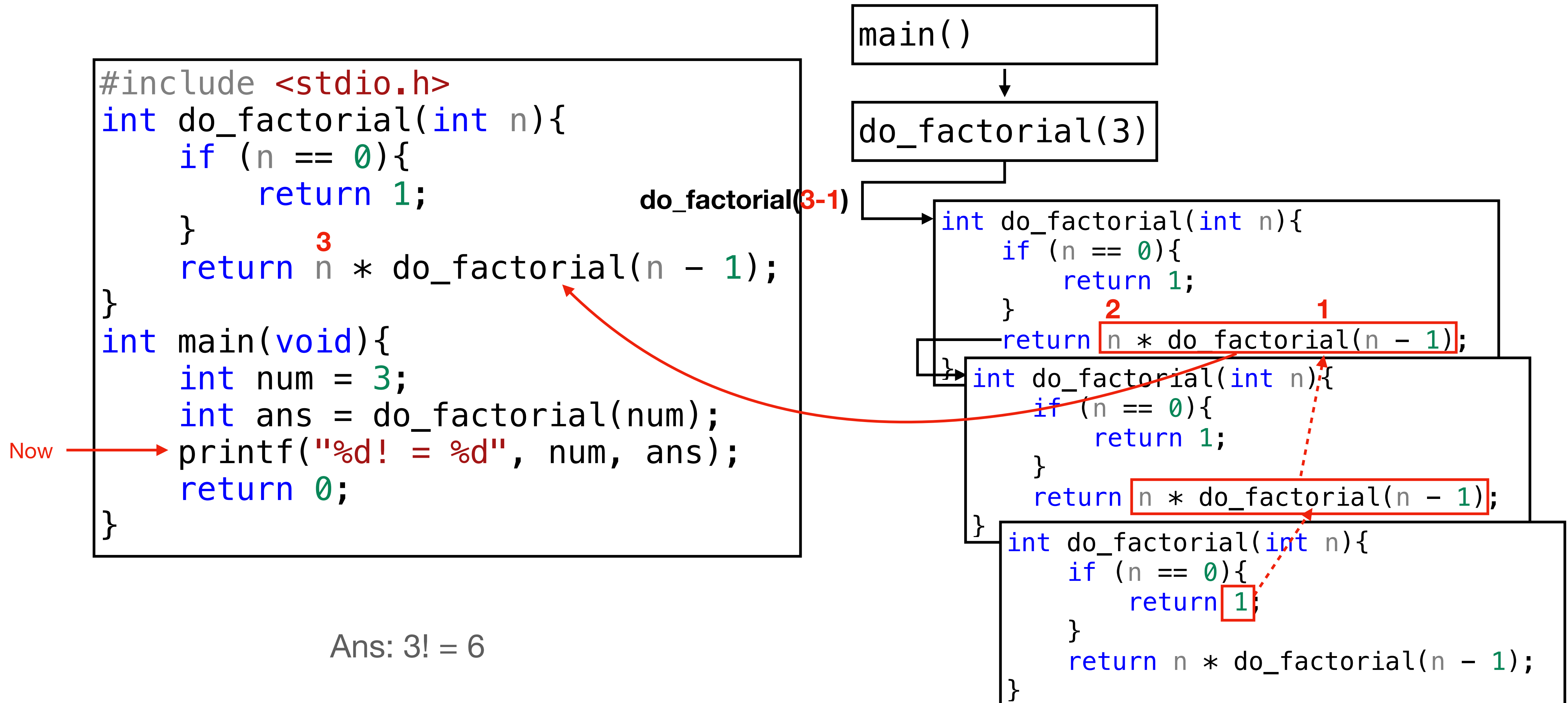
[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



[Illustration] Factorial

C-course-materials/05-Functions/factorial.c



Greatest Common Divisor (GCD)

C-course-materials/03-Program_control_while/while_gcd.c

```
#include <stdio.h>
int main(){
    int a, b, temp;
    scanf("%d %d", &a, &b);
    while (b != 0) {
        temp = a % b;
        a = b;
        b = temp;
    }
    printf("最大公因數為%d", a);
}
```

Your turn

Summary of Recursion

- Recursion is not an algorithm. Instead, it is a technique to solve a big question from smaller pieces.

Lifetime of C Variables

Outline

- Local variable (區域變數)
- Global variable (全域變數)
- Static variable (靜態變數)

[Illustration] Local vs. Global variables

C-course-materials/compare_local_global.c

Global
variable



Local
variable



```
#include <stdio.h>
int globalVar = 100;

void do_print(void){
    printf("Global variable: %d\n", globalVar);
    // printf("Local variable: %d\n", localVar); // will cause an error
}

int main(void){
    int localVar = 0;
    printf("Global variable: %d\n", globalVar);
    do_print();
}
```

[Definition] Global variable

- The declaration of a global variable is **outside any function** in a program.
- In this way, all functions or code blocks in a program can use the global variable.

Scope of a Global Variable

C-course-materials/compare_local_global.c

Scope:
the whole
program

```
#include <stdio.h>
int globalVar = 100;

void do_print(void){
    printf("Global variable: %d\n", globalVar);
    // printf("Local variable: %d\n", localVar); // will cause an error
}

int main(void){
    int localVar = 0;
    printf("Global variable: %d\n", globalVar);
    do_print();
}
```

[Definition] Local Variable

- Declaring a variable inside a function definition (including the main function) makes the variable name **local** to the code block.
- Life of a local variable:
 - Each variable's storage exists only from the declaration to the end of the block
 - Execution of the declaration allocates the storage, computes the initial value, and stores it in the variable. The end of the block deallocates the storage.

Scope of a Local Variable (1)

C-course-materials/compare_local_global.c

```
#include <stdio.h>
int globalVar = 100;

void do_print(void){
    printf("Global variable: %d\n", globalVar);
    // printf("Local variable: %d\n", localVar); // will cause an error
}

int main(void){
    int localVar = 0;
    printf("Global variable: %d\n", globalVar);
    do_print();
}
```

Scope:
within the
function



Scope of a Local Variable (2)

C-course-materials/local_var_scope.c

```
#include <stdio.h>
int do_factorial(int n){
    int i, total = 1;
    for (i = 1; i <= n; i++){
        total *= i;
    }
    return total;
}
int main(void){
    int ans;
    ans = do_factorial(5);
    printf("Factorial(5): %d", ans);
    return 0;
}
```

Scope of **n**

Scope of **i**, **total**

Scope of **ans**

[Definition] static variable

- The declaration of a global variable is **outside any function** in a program.
- In this way, all functions or code blocks in a program can use the global variable.

Example to use a static local variable

C-course-materials/05-Functions/static_local.c

`sum` is a **static local** variable.

```
#include <stdio.h>
int add(int a, int b){
    static int sum = 0;
    sum += (a + b);
    return sum;
}
int main(void){
    int num_a = 5;
    int num_b = 6;
    int result;
    for (int i = 0; i < 5; i++){
        result = add(num_a, num_b);
        printf("Sum: %d\n", result);
    }
}
```

`sum` is a **local** variable.

```
#include <stdio.h>
int add(int a, int b){
    int sum = 0;
    sum += (a + b);
    return sum;
}
int main(void){
    int num_a = 5;
    int num_b = 6;
    int result;
    for (int i = 0; i < 5; i++){
        result = add(num_a, num_b);
        printf("Sum: %d\n", result);
    }
}
```


[Notes] static variable

The **static** keyword can **control**:

1. Life cycle of a variable

- A `static` variable declared inside a function retains its value across function calls and remains in memory until the program ends.

2. External Linkage

- A `static` **global** variable or function is accessible only within the file where it is declared (often called “file scope”).
- (We will not go into detail on this today.)