Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 5

Instructor Younghoon Kim

Recap of Recap

SUNG KYUN KWAN UNIVERSITY(SKKU)

- **Control flow**
- **Branch**
  - if, if…else, switch
- **Loop**
  - while, do…while, for

Recap

- **Functions**
  - Definition: return type, function name, parameter list
  - How to call a function
    - » with or without parameters
    - » usages of a return value

- **Remember the 'divisor war' example.**

Today

- Global Variable
- Recursion

Scoping (Global Variable)

Scoping

- **Types of variables regarding scopes**
  - Global variables
  - Local variables

- **Types of variables regarding allocated memory**
  - Static variables
    - » A variable that has been allocated "statically", and its lifetime is the entire run of the program.
    - » Although the concept is very important, it will be covered very briefly in this course.
  - Automatic or dynamic variables

Variable Scope

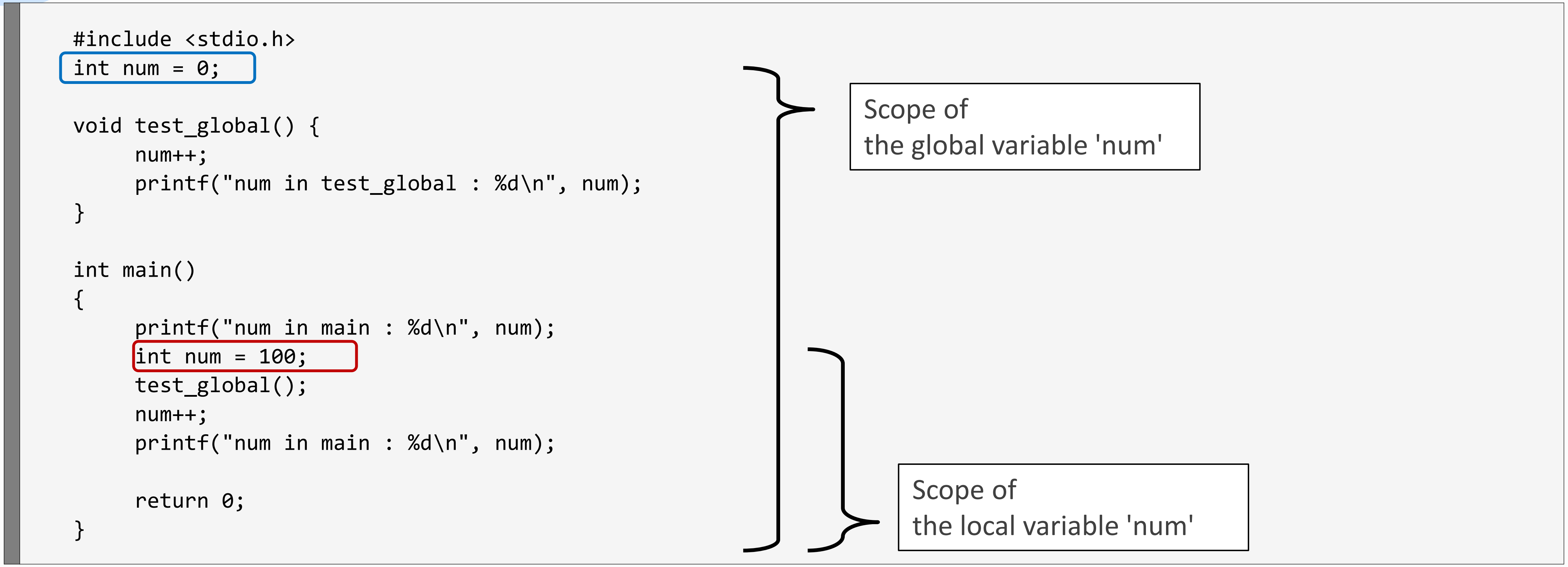
- **Local Variables**
  - Most of variables you are using and will use are ***locally*** valid.
  - The term ***'locally'*** usually and roughly means ***inside of a block.***

- **Global Variables**
  - Variables which are declared outside of all blocks

```
int global_variable = 100;                    // global variable

int main(void)
{
     int local_variable = 100;                // local variable
     ...
}
```

Global Variables



```
#include <stdio.h>
int num = 0;

void test_global() {
     num++;
     printf("num in test_global : %d\n", num);
}

int main()
{
     printf("num in main : %d\n", num);
     int num = 100;
     test_global();
     num++;
     printf("num in main : %d\n", num);

     return 0;
}
```

Scope of the global variable 'num'
Scope of the local variable 'num'

- **Global variables live until the program exits.**
- **Local variables with the same names hide global variables.**
  - To get the value of a variable, a program looks for the name from closer blocks.
  - If there is none, it checks the global variables with that name.

Global Variable Example

DEMO

- **Counting the number of function calls**
  - One typical usage of global variables.

```
#include <stdio.h>

int num = 0;                     // Declaring and initializing a global variable

void test_global() {
    num++;
    printf("number that test_global has been called : %d\n", num);
}

int main() {
    for(int i=0; i<10; i++)
         test_global();
    return 0;
}
```

Varying variable scopes

DEMO

- **Variables with various scopes**
  - Moving or erasing braces will have different results.

```
#include <stdio.h>
int num = 0;
void test_global() {
     printf("num in test_global : %d\n", ++num);
}

int main() {
    printf("num in main : %d\n", num);
    {
         int num = 100;
         test_global();
         test_global();
         printf("num in main : %d\n", ++num);
    }
    printf("num in main : %d\n", num);
    return 0;
}
```

Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

Programming Basics 5

Instructor Younghoon Kim

Recursion 1

Recursion

- A function is **recursive** if it **calls itself**.

```
// Wrong and naïve example of a recursive function
void Recursive(void)
{
    Recursive();
}
```

- CHECK: What happens with the above function?
  - » When the function is called, it will call itself infinitely.
  - » It needs to stop at some points to prevent a runtime error.

- **Recursion is useful for some types of problems.**

Recursive Example with Factorial

- **The factorial of a nonnegative integer n, n!:**

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1,$$

where 1! = 1 and 0! = 1.

(Ex: 5! = 5 * 4 * 3 * 2 * 1 = 120)

- **An iterative way of calculating the factorial:**

```
factorial = 1;
for (counter = number; counter >= 1;counter--)
   factorial *= counter;
```

Recursive Example with Factorial

- **A recursive definition of the factorial function:**

$$n! = n \cdot (n - 1)!$$

$$(\text{Ex: } 5! = 5 \cdot 4!\ )$$

- **A recursive way of calculating the factorial:**

```
return number * factorial(number - 1);
```

- under the assumption that the function `factorial` calculates a factorial number for an integer input
- The call factorial(number - 1) is a slightly simpler problem than the original calculation factorial(number).
- Eventually, the function will be called with one, which is the base case.

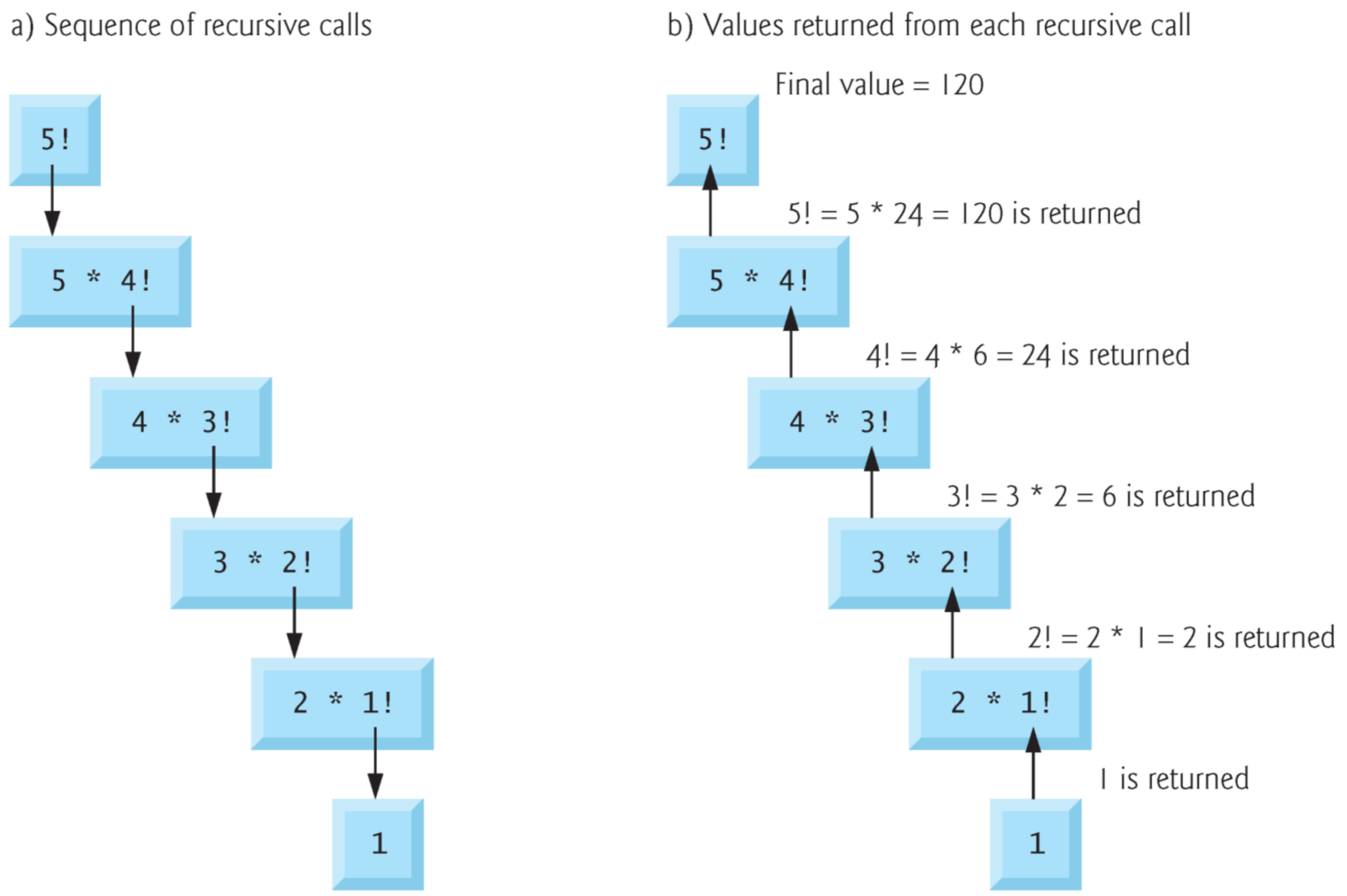
Recursive calls

- **Recursion call (Recursion step)**
  - Function calls that calls itself to work on the same but smaller problem.
  - Returning the result is done by combining the current result with the result of the smaller problem.

- **Base case**
  - Simplest case(s) for the recursive calls
  - When called with a base case, the function simply returns a result.

- **Termination of recursion**
  - When the function is called with the base condition, there will be no further recursive calls but just returning all the way back to the original function call.
  - Reducing problems by dealing smaller problems is a key idea.

The evaluation of 5!


a) Sequence of recursive calls
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

Recursively Calculating Factorials

DEMO

- **Variations on the data types will be shown in the demo.**
  - In fact, the following code is not working properly due to the data size.
  - An iterative implementation will also be shown.

```c
#include <stdio.h>

int factorial(int num) {
  if (num <= 1)
    return 1;
  else
    return num * factorial(num - 1);
}

int main() {
  for(int num = 0; num <= 21; num++)
    printf("%d! is %d\n", num, factorial(num));
  return 0;
}
```

Naive Recursive Function (Revisited)

DEMO

```
#include <stdio.h>

void Recursive(void)
{
        printf("Recursive Call!\n");
        Recursive();
}

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

- Calling itself inside of a function .
- When does it end calling itself?

Recursion Format

DEMO

```c
#include <stdio.h>

int Recursive(int n)
{
    if (n == 0)
        return 0;
    printf("Recursive Call! (%d)\n", n);
    Recursive(n-1);
}

int main(void)
{
    Recursive(10);
    return 0;
}
```

- **You must specify termination cases.**

Exercise: Sum Function

DEMO

- **Make a function that returns the sum of numbers in [1..num].**
  - In an iterative way or a recursive way

```
#include <stdio.h>

int sum(int num) {

/* Fill in here */

}

int main() {
  int num;
  printf("Input a number: ");
  scanf("%d", &num);
  printf("Sum %d (inclusive) is %d\n", num, sum(num));

  return 0;
}
```

Sum Function (Iterative)

DEMO

- **Make a function that returns the sum of numbers in [1..num].**
  - Tracing with printf will be shown

```c
#include <stdio.h>

int sum(int num) {
  int ret = 0;
  for(int i=1; i<=num; i++) {
    ret += i;
  }
  return ret;
}

int main() {
  int num;
  printf("Input a number: ");
  scanf("%d", &num);
  printf("Sum %d (inclusive) is %d\n", num, sum(num));

  return 0;
}
```

Sum Function (Recursive)

DEMO

- **Make a function that returns the sum of numbers in [1..num].**
  - Tracing with printf will be shown

```c
#include <stdio.h>

int sum_recursion(int num)
{
  if (num == 0)
    return 0;
  return num + sum_recursion(num-1);
}

int main()
{
  int num;
  printf("Input a number: ");
  scanf("%d", &num);
  printf("Sum %d (inclusive) is %d\n", num, sum_recursion(num));

  return 0;
}
```

Reading Materials

Recursion

- The programs we’ve dealt are generally structured as functions that call one another in a disciplined, hierarchical manner. For some types of problems, it’s useful to have functions call themselves. A recursive function is a function that calls itself either directly or indirectly through another function. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem.

Recursion

- The function actually knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a recursive call or the recursion step.
- The recursion step also includes the keyword return, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

SUNG KYUN KWAN UNIVERSITY(SKKU)

Recursion

- The recursion step executes while the original call to the function is paused, waiting for the result from the recursion step. The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it’s called with into two conceptual pieces.

- For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case. When the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to main.

Basis and Practice in Programming

SUNG KYUN KWAN UNIVERSITY(SKKU)

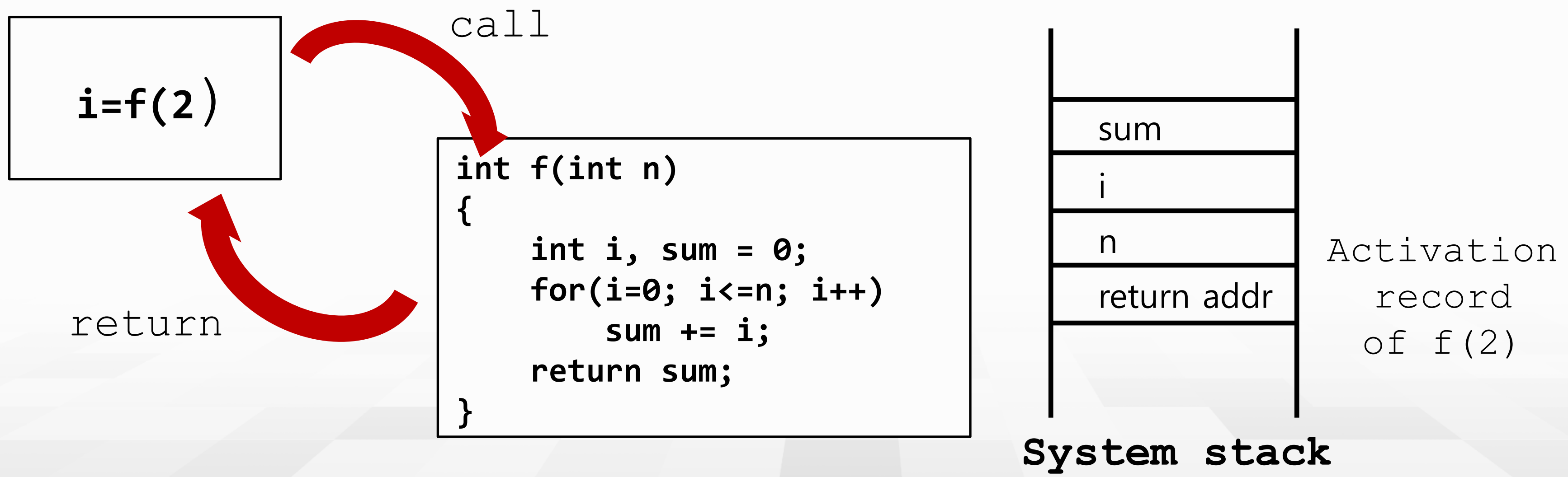
Programming Basics 5

Instructor Younghoon Kim

Recursion 2

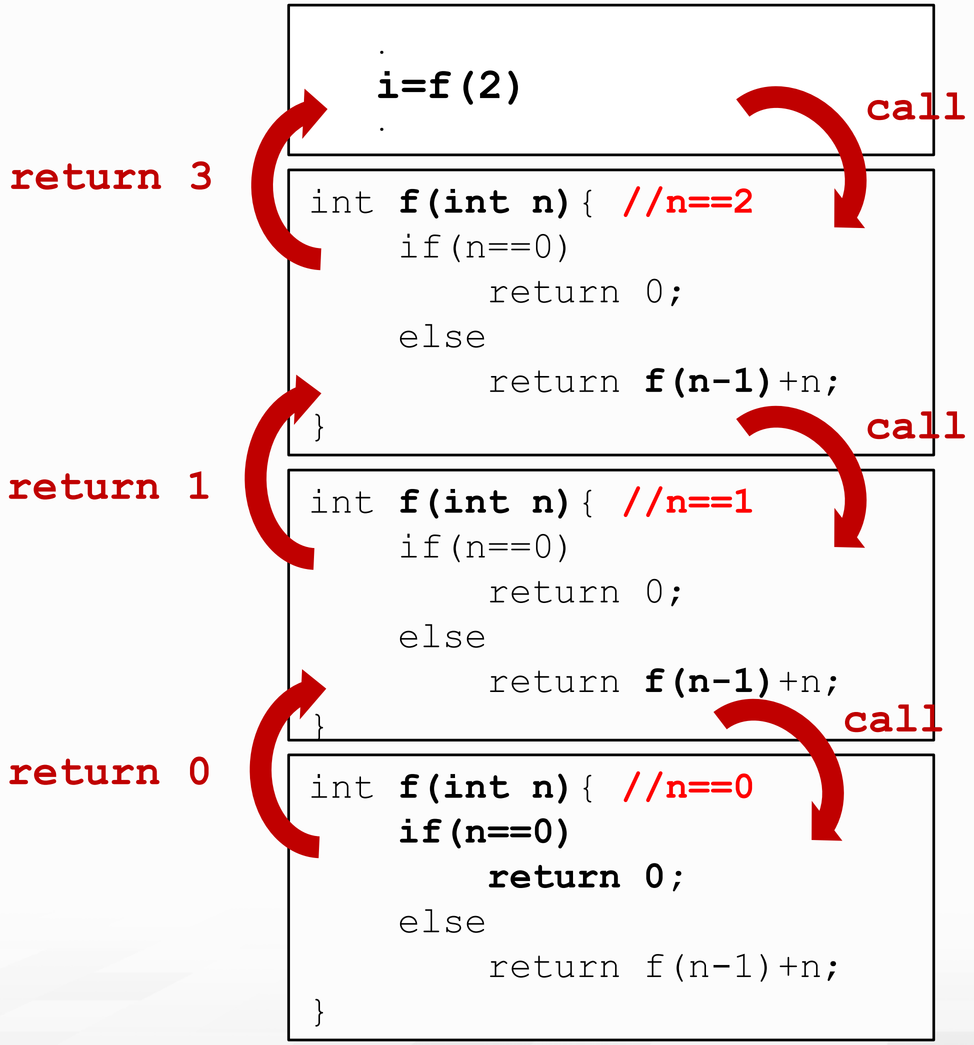
Function call and return - The memory point of view

- **When called**
  - Return address is kept into system stack
  - All local variables are newly allocated into system stack
- **When returning**
  - All local variables are removed
  - Returning to the address kept in the stack



```
int f(int n)
{
    int i, sum = 0;
    for(i=0; i<=n; i++)
        sum += i;
    return sum;
}
```

Function call and return - The memory point of view

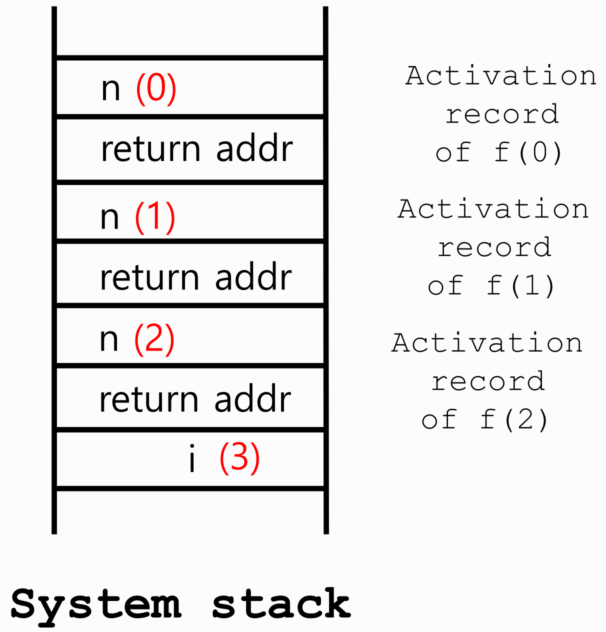


```
   .
   i=f(2)
   .
```

```
int f(int n){ //n==2
    if(n==0)
        return 0;
    else
        return f(n-1)+n;
}
```

```
int f(int n){ //n==1
    if(n==0)
        return 0;
    else
        return f(n-1)+n;
}
```

```
int f(int n){ //n==0
    if(n==0)
        return 0;
    else
        return f(n-1)+n;
}
```


n (0)
return addr
n (1)
return addr
n (2)
return addr
i (3)
Activation record of f(0)
Activation record of f(1)
Activation record of f(2)

System stack

Iteration vs. Recursion

$$sum(n) = \sum_{k=1}^{n} k$$

$$sum(n) = \begin{cases} 1 & , n = 1 \\ n + sum(n-1) & , n > 1 \end{cases}$$

```
/* Iterative version */
int sum ( int n )
{
    int sum = 0, k;

    for( k = 1 ; k <=n ; k++ )
         sum += k;

    return sum;
}
```

```
/* Recursive version */
int sum ( int n )
{
     if ( n == 1 )
          return 1;
     else
          return n + sum( n-1 );
}
```

Iteration vs. Recursion (cont.)

- **Iterative algorithm**
  - May be more efficient
  - No additional function calls
  - Run faster, use less memory

- **Recursive algorithm**
  - Relatively Higher overhead
  - Many function calls / returns : expensive in time
  - Activation records for each call : more memory
  - May be simpler and easy to understand

Recursion Example - Fibonacci sequence

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
```

```
Fibonacci(1) = 0
Fibonacci(2) = 1
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2), when n>=2
```

- **Function design**
  - Return type: integer
  - Function name: Fibonacci
  - Parameter: one integer
- **Recursion design (Let n be the parameter name.)**
  - Return value: Fibonacci(n-1) + Fibonacci(n-2)
  - Termination cases: it stops calling when n is 0 or 1.

Fibonacci Sequence

DEMO

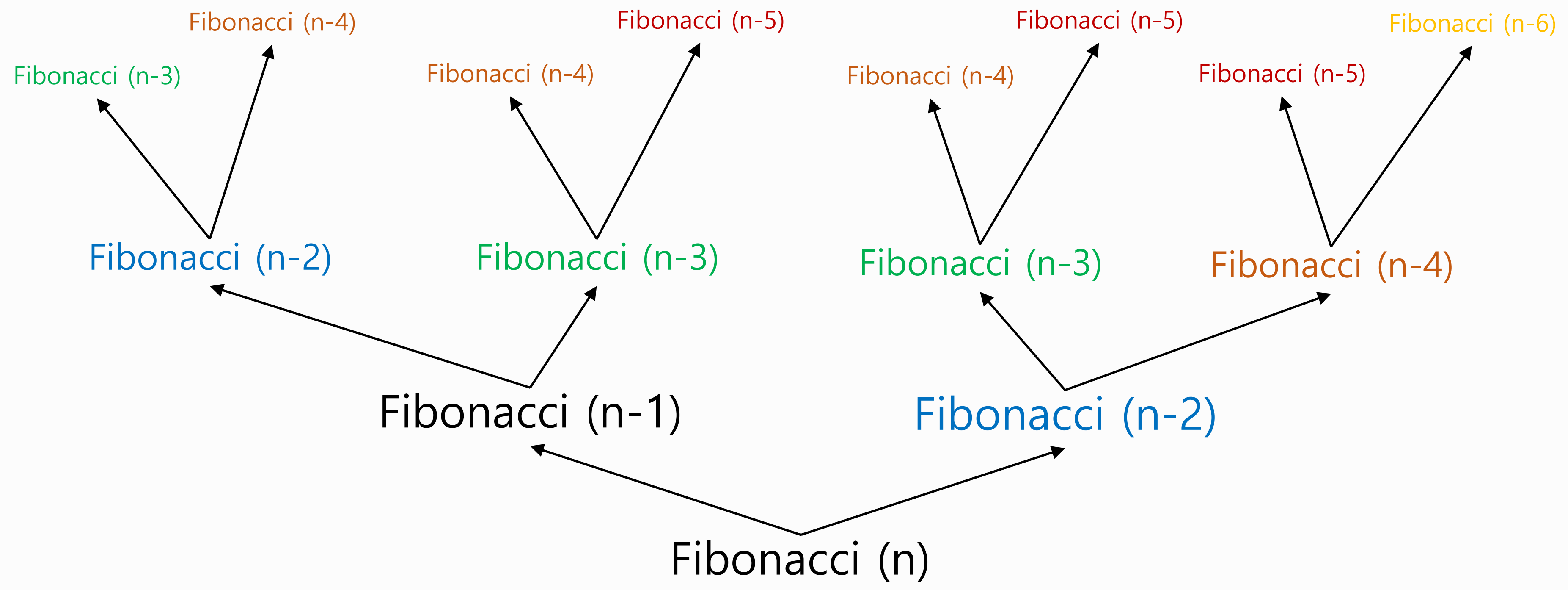
```c
#include <stdio.h>

int Fibonacci(int num)
{
  if (num == 1)
    return 0;
  if (num == 2)
    return 1;
  return Fibonacci(num - 2) + Fibonacci(num - 1);
}

int main()
{
  int num;
  printf("Input a number: ");
  scanf("%d", &num);
  printf("%dth Fibonacci number is %d.\n", num, Fibonacci(num));

  return 0;
}
```

An issue on recursive version



- **We may reuse results from former function calls.**

"memoization"

Fibonacci sequence - Iteration version

SUNG KYUN KWAN UNIVERSITY(SKKU)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- **Remind the Fibonacci sequence.**
- **Our concern is limited to only three recent numbers.**
  - (0, 1, 1), (1, 1, 2), (1, 2, 3), (2, 3, 5), …
- **So, how can we make an iteration version of Fibonacci function?**
  - A loop will be used.

Fibonacci sequence - Iteration version

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- **Function design (same with recursive version)**
  - Return type: integer
  - Function name: Fibonacci
  - Parameter: one integer

- **Iteration design (Let n be the parameter name.)**
  - Temporal variables for recent three numbers (num1, num2, num3)
  - Return value: num3
  - Number of loops: n – 3 (Why??)
    - It can be different based on your design.

Fibonacci sequence - Iteration version

SUNG KYUN KWAN UNIVERSITY(SKKU)

- **Designing inside of a loop**

```
// Pseudo code
num1: 0, num2: 1, num3: 1 // initial values
loop for moving window
    num1 ← num2
    num2 ← num3
    num3 ← num1 + num2
for ends
return num3
```

loop number	window (num1, num2, num3)
1st loop	0, **1, 1, 2,** 3, 5, 8, …
2nd loop	0, 1, **1, 2, 3,** 5, 8, …
3rd loop	0, 1, 1, **2, 3, 5,** 8, …
4th loop	0, 1, 1, 2, **3, 5, 8,** …
5th loop	0, 1, 1, 2, 3, **5, 8,** …

Fibonacci Sequence (Iterative)

DEMO

- Execution times of both style will be compared in this demo.

```c
#include <stdio.h>

int Fibonacci(int num) {
  if (num == 1) return 0;
  if (num == 2) return 1;
  if (num == 3) return 1;

  int num1 = 0, num2 = 1, num3 = 1;

  for(int i=0; i<num-3; i++) {
    num1 = num2;
    num2 = num3;
    num3 = num1 + num2;
  }
  return num3;
}

// main is the same with above demo
```

Performance Comparison Iteration vs. Recursion

SUNG KYUN KWAN UNIVERSITY(SKKU)

- **Iterative algorithm**
  - May be more efficient
  - No additional function calls
  - **Run faster, use less memory**

- **Recursive algorithm**
  - Relatively Higher overhead
  - **Many function calls / returns : expensive in time**
  - Activation records for each call : more memory
  - May be simpler and easy to understand

Exponential Complexity

- Each level of recursion in the fibonacci function has a doubling effect on the number of calls—the number of recursive calls that will be executed to calculate the nth Fibonacci number is on the order of 2n. This rapidly gets out of hand.

- Calculating only the 20th Fibonacci number would require on the order of 220 or about a million calls, calculating the 30th Fibonacci number would require on the order of 230 or about a billion calls, and so on.

- Computer scientists refer to this as exponential complexity. Problems of this nature humble even the world's most powerful computers! The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches.