Computer Programming & Problem Solving CS100

Mrs Sanga G. Chaki

Department of Computer Science and Engineering

National Institute of Technology, Goa

November, 2022

Functions

1. A function is a self-contained block of statements that perform a specific, well defined task.
   a) a task that is always performed exactly in the same way
2. Example
   1. A function to add two numbers
   2. A function to find the largest of n numbers
3. A function will carry out its intended task whenever it is **called or invoked**
   1. Can be called multiple times

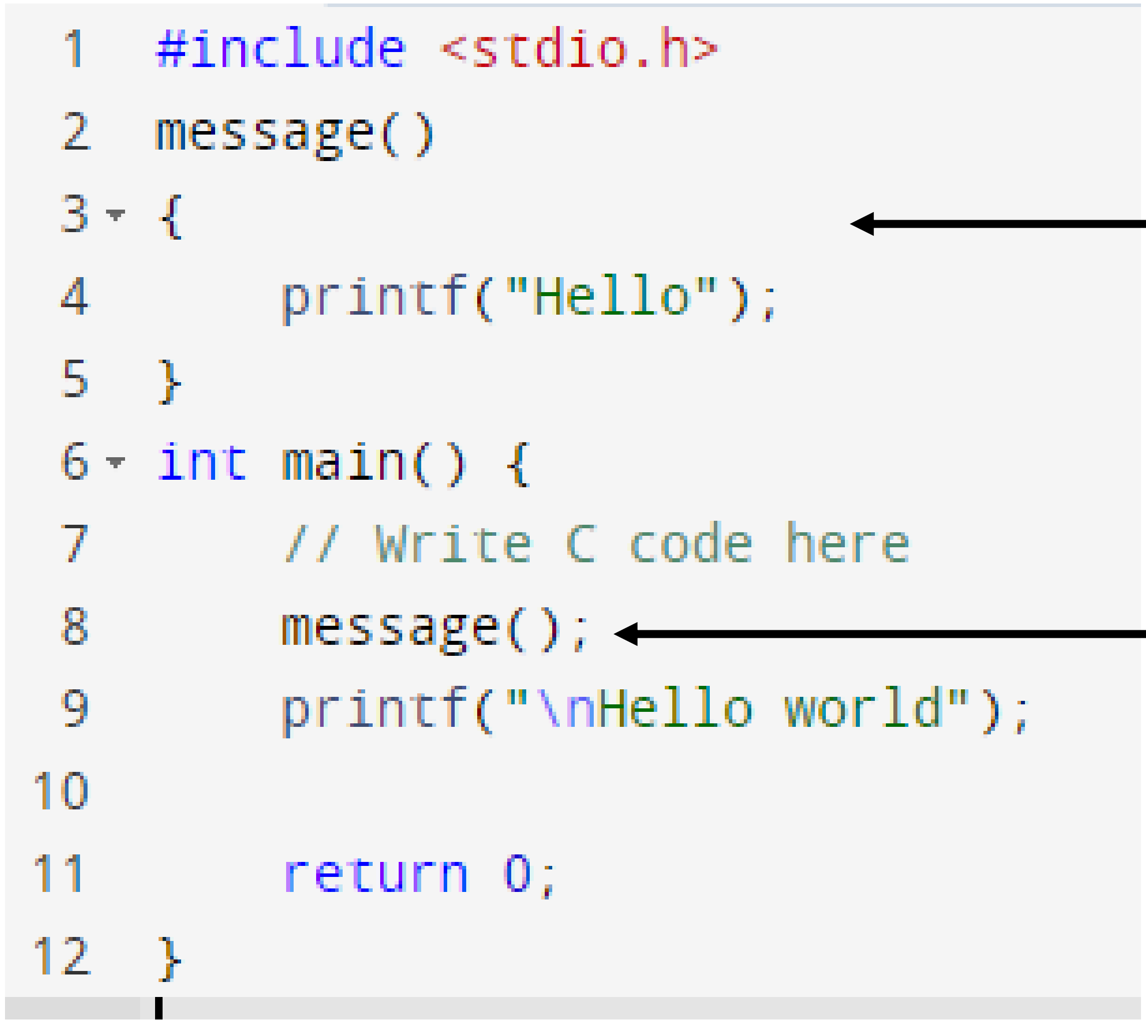
Function – Why Use Functions

1. Re-use: avoids rewriting the same code over and over.
2. Divide-and-conquer approach: easier to write programs and keep track of what they are doing

Functions

1. Every C program consists of one or more functions.
2. One of these functions must be called “main”.
3. Execution of the program always begins by carrying out the instructions in “main”.
4. Functions may call other functions (or itself) as instructions

Function – first example



```c
#include <stdio.h>
message()
{
    printf("Hello");
}
int main() {
    // Write C code here
    message();
    printf("\nHello world");

    return 0;
}
```

Function Definition – First Line, function body

Function Call

Function – Control Flow

1. What is happening here?
2. Here, main( ) itself is a function and through it we are calling the function message( ).
3. What do we mean when we say that main( ) calls the function message( )?
4. We mean that the control passes to the function message( ).
5. The activity of main( ) is temporarily suspended; it falls asleep while the message( ) function wakes up and goes to work.
6. When the message( ) function runs out of statements to execute, the control returns to main( ), which comes to life again and begins executing its code at the exact point where it left off.
7. Thus, main( ) becomes the ‘calling’ function, whereas message( ) becomes the ‘called’ function.

Multiple Functions – Control Flow

```
italy( )
{
      printf ( "\nI am in italy" ) ;

}
brazil( )
{
      printf ( "\nI am in brazil" ) ;
}
argentina( )
{
      printf ( "\nI am in argentina" ) ;
}
```

```
main( )
{
      printf ( "\nI am in main" ) ;
      italy( ) ;
      brazil( ) ;
      argentina( ) ;
}
```

What if I change the calling order?

OUTPUT

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

Function – Basic Rules

1. Any C program contains at least one function.
2. If a program contains only one function, it must be main( ).
3. If a C program contains more than one function, then one (and only one) of these functions must be main( ), because program execution always begins with main( ).
4. There is no limit on the number of functions that might be present in a C program.
5. Each function in a program is called in the sequence specified by the function calls in main( ).
6. After each function has performed its task, control returns to main().
7. When main( ) runs out of function calls, the program ends.

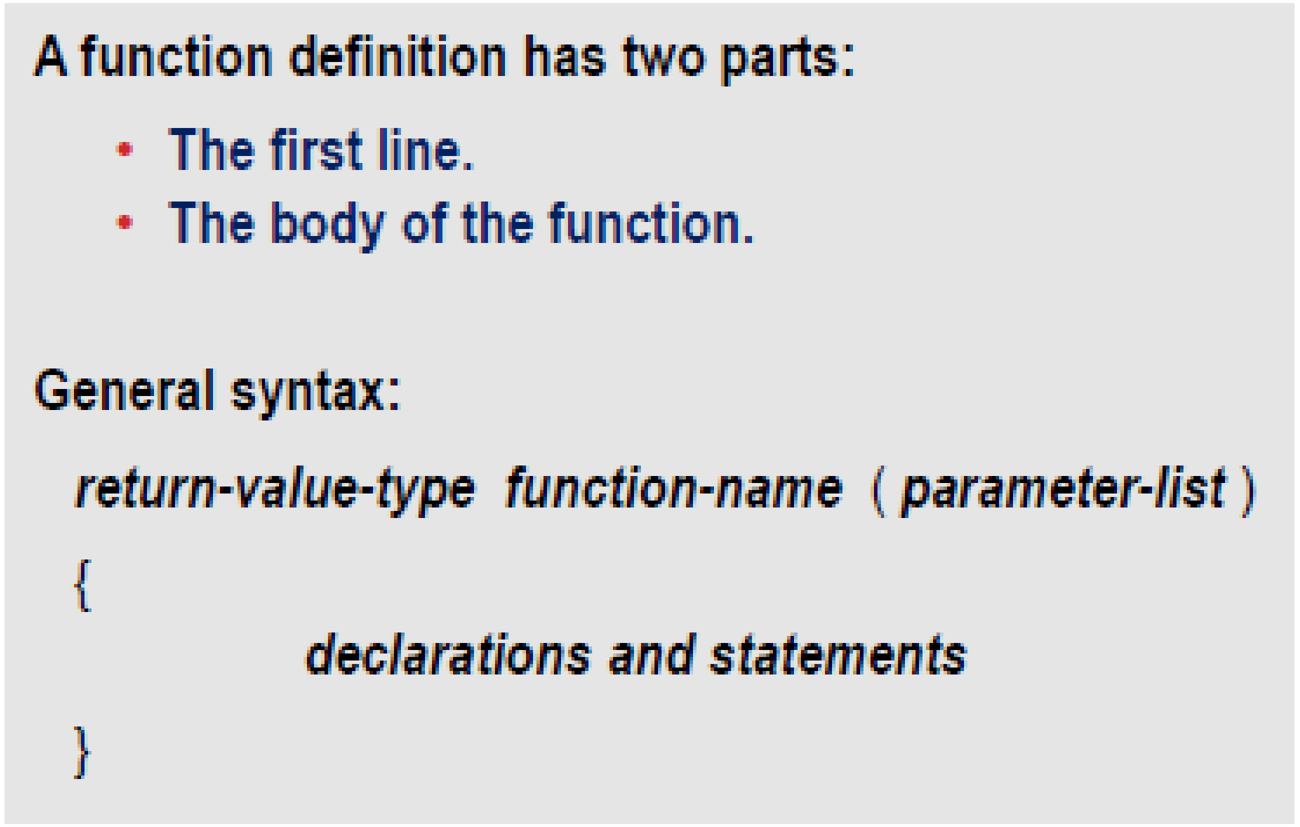
Function – Types

1. Library functions Ex. printf( ), scanf( ) etc.
2. User-defined functions Ex. argentina( ), brazil( ) etc.

Function – New Terms

1. Function Calling
2. Function Declaration
3. Function Definition

Defining a Function



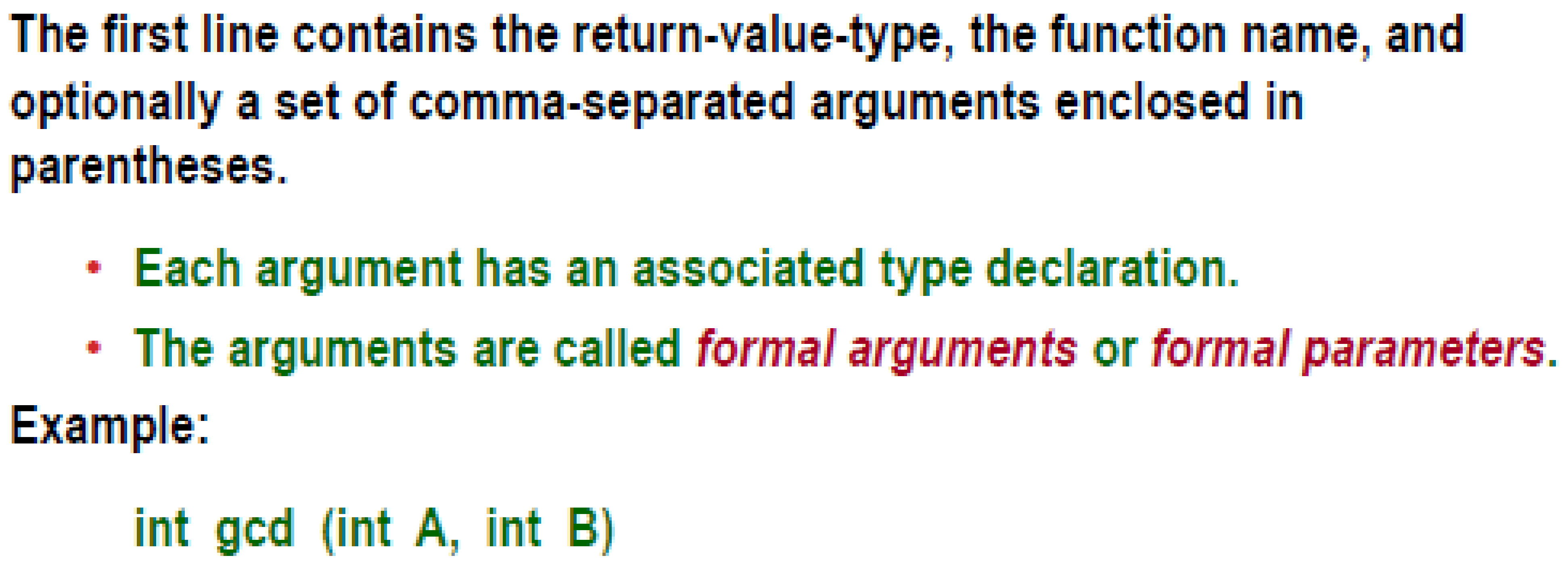
A function definition has two parts:

- The first line.
- The body of the function.

General syntax:

```
  return-value-type  function-name  ( parameter-list )
  {
              declarations and statements
  }
```

Defining a Function



The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.

- Each argument has an associated type declaration.
- The arguments are called *formal arguments* or *formal parameters*.

Example:

```
int gcd (int A, int B)
```

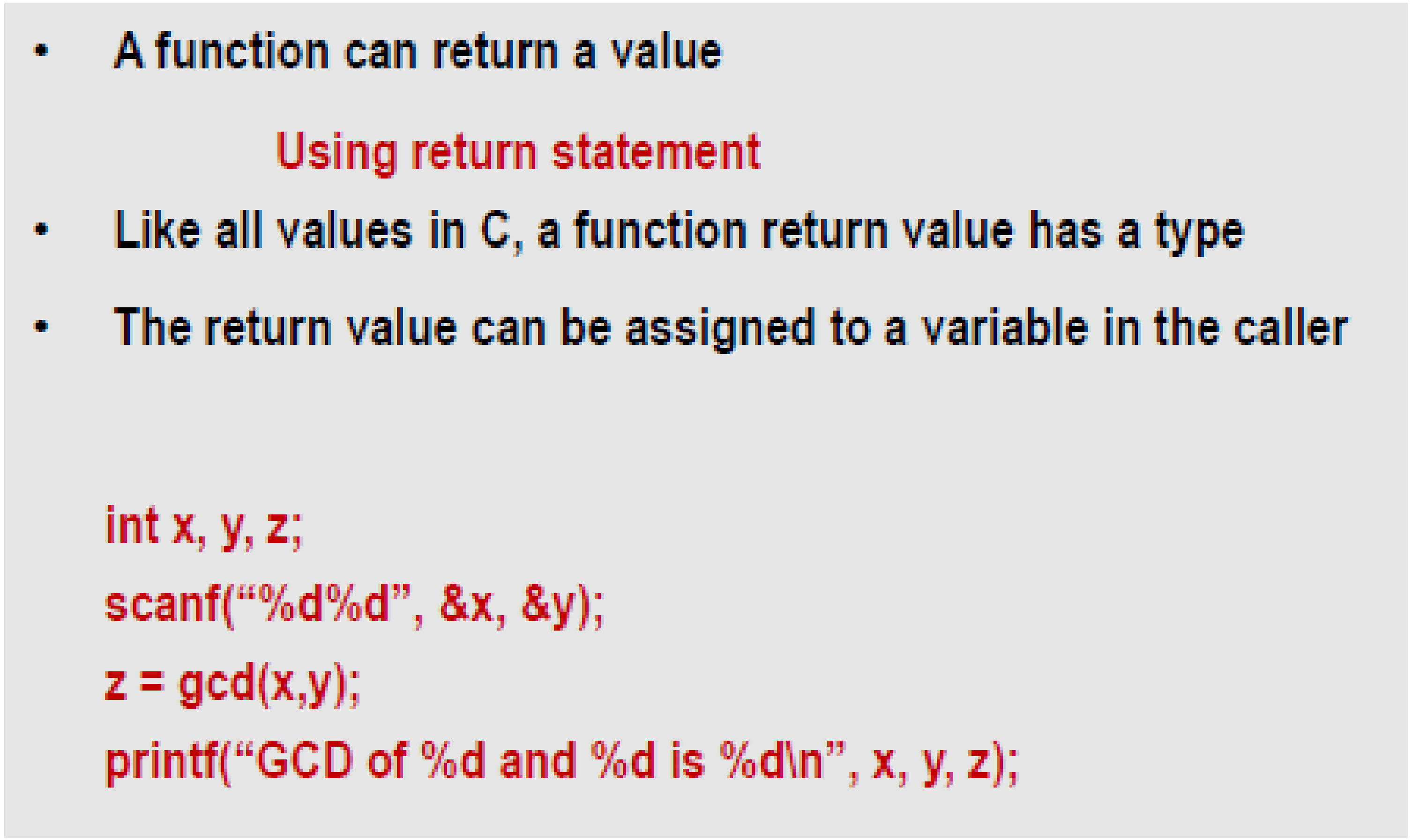
```
int sum(int A, int B)
```

1. The mechanism used to convey information to the function is the ‘argument’/parameter

Defining a Function - Example

```
int sum(int A, int B)
{
        int sum;
        sum = A+B;
        return (sum);
}
```

Return Value



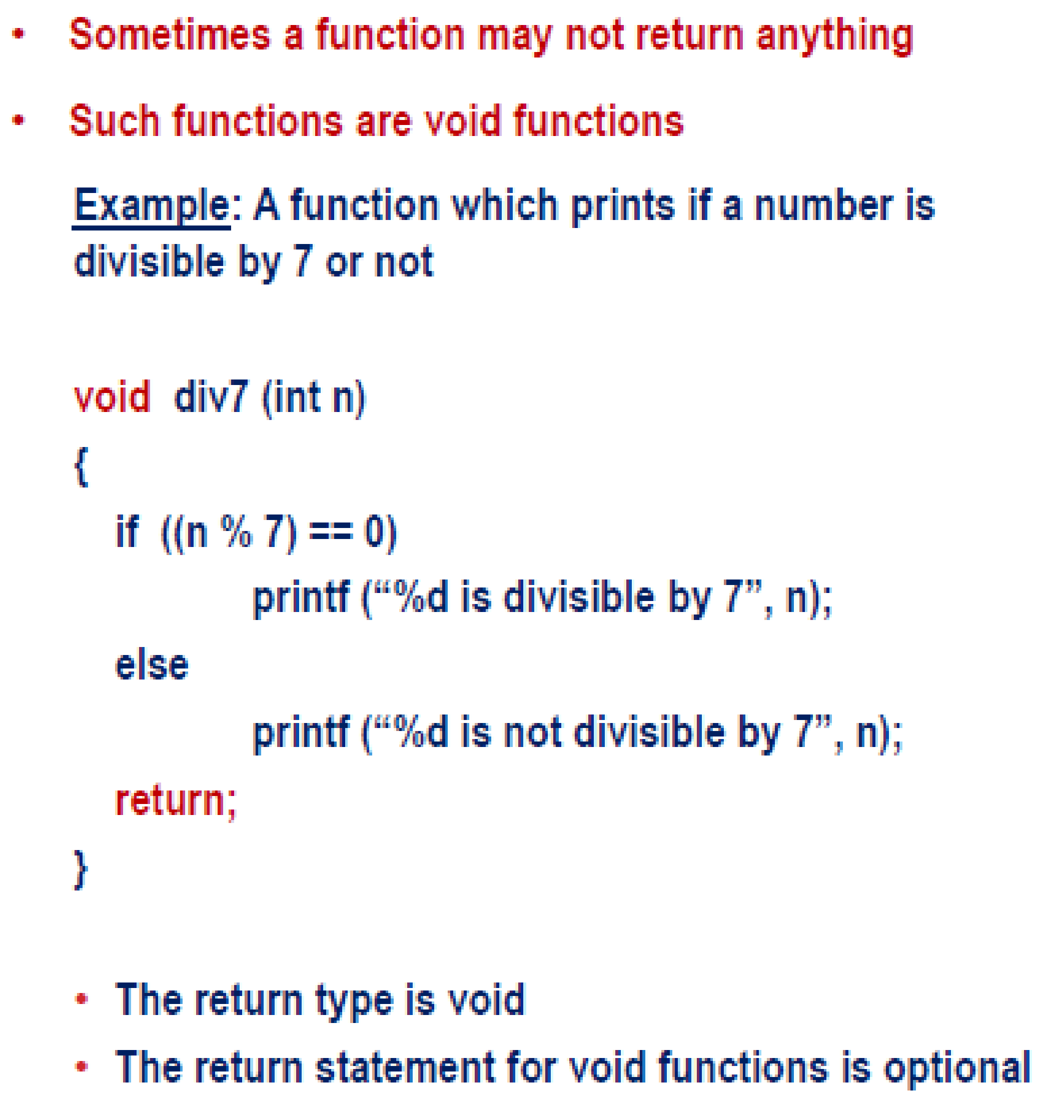
- A function can return a value

  Using return statement

- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x,y);
printf("GCD of %d and %d is %d\n", x, y, z);
```

Return Value



- Sometimes a function may not return anything
- Such functions are void functions

Example: A function which prints if a number is divisible by 7 or not

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
            printf ("%d is divisible by 7", n);
   else
            printf ("%d is not divisible by 7", n);
   return;
}
```

- The return type is void
- The return statement for void functions is optional

```
void sum(int A, int B)
{
        int sum;
        sum = A+B;
        printf("sum = %d",sum);
        //return (sum);
}
```

The Return statement

In a value-returning function (result type is not void), return does two distinct things

- specify the value returned by the execution of the function
- terminate that execution of the callee and transfer control back to the caller

A function can only return one value

- The value can be any expression matching the return type
- but it might contain more than one return statement.

In a void function

- return is optional at the end of the function body.
- return may also be used to terminate execution of the function explicitly before reaching the end.
- No return value should appear following return.

The Return statement

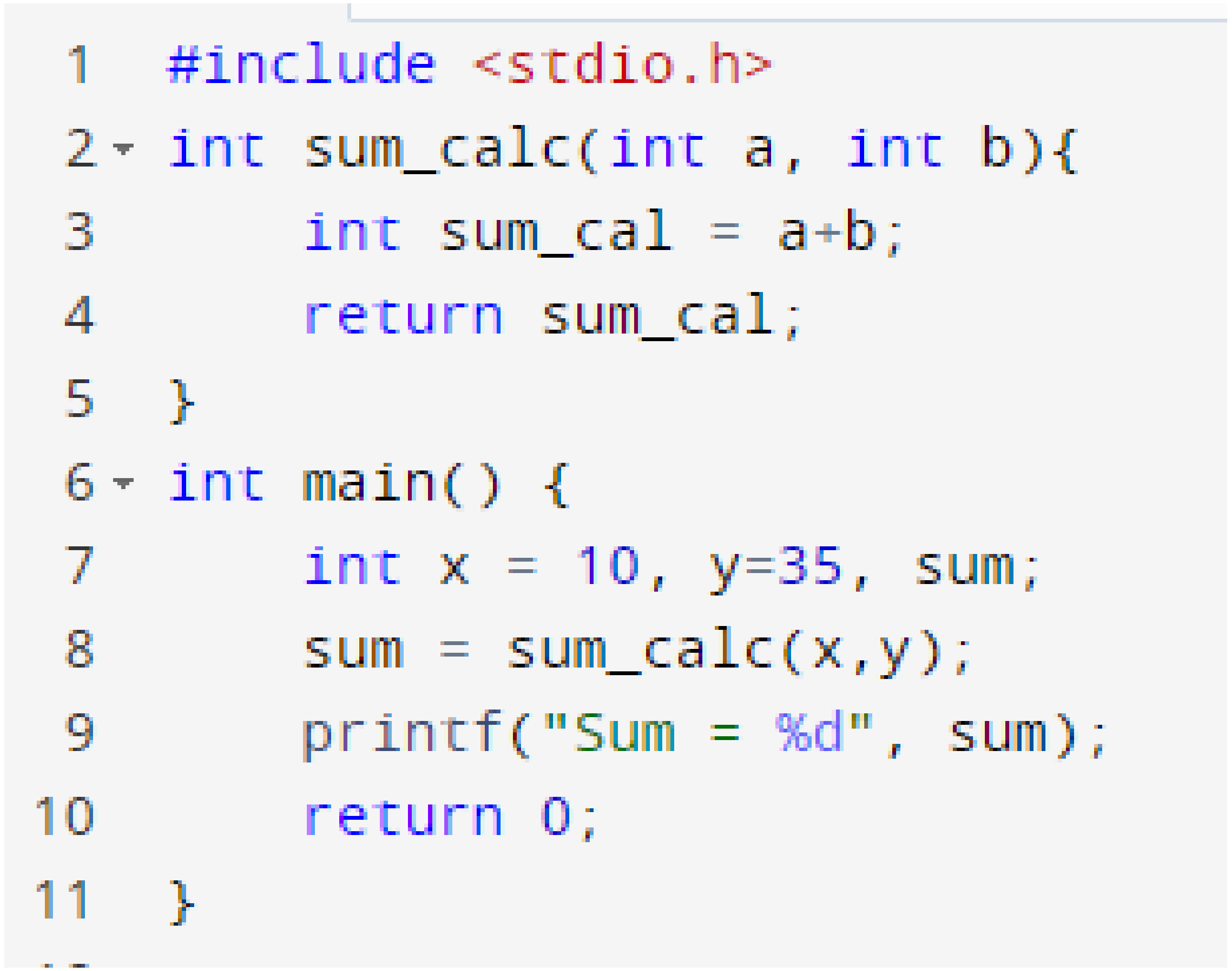
```
void compute_and_print_itax ()
{
   float income;
   scanf (“%f”, &income);
   if (income < 50000)   {
               printf (“Income tax = Nil\n”);
               return;  /* Terminates function execution */
   }
   if (income < 60000)   {
               printf (“Income tax = %f\n”, 0.1*(income-50000));
               return;  /* Terminates function execution */
   }
   if (income < 150000) {
                printf (“Income tax = %f\n”, 0.2*(income-60000)+1000);
               return ; /* Terminates function execution */
   }
   printf (“Income tax = %f\n”, 0.3*(income-150000)+19000);
}
```

Calling a function - Rules

- Called by specifying the function name and parameters in an instruction in the calling function

- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameter in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

Calling a function – Example – Actual and Formal Parameters

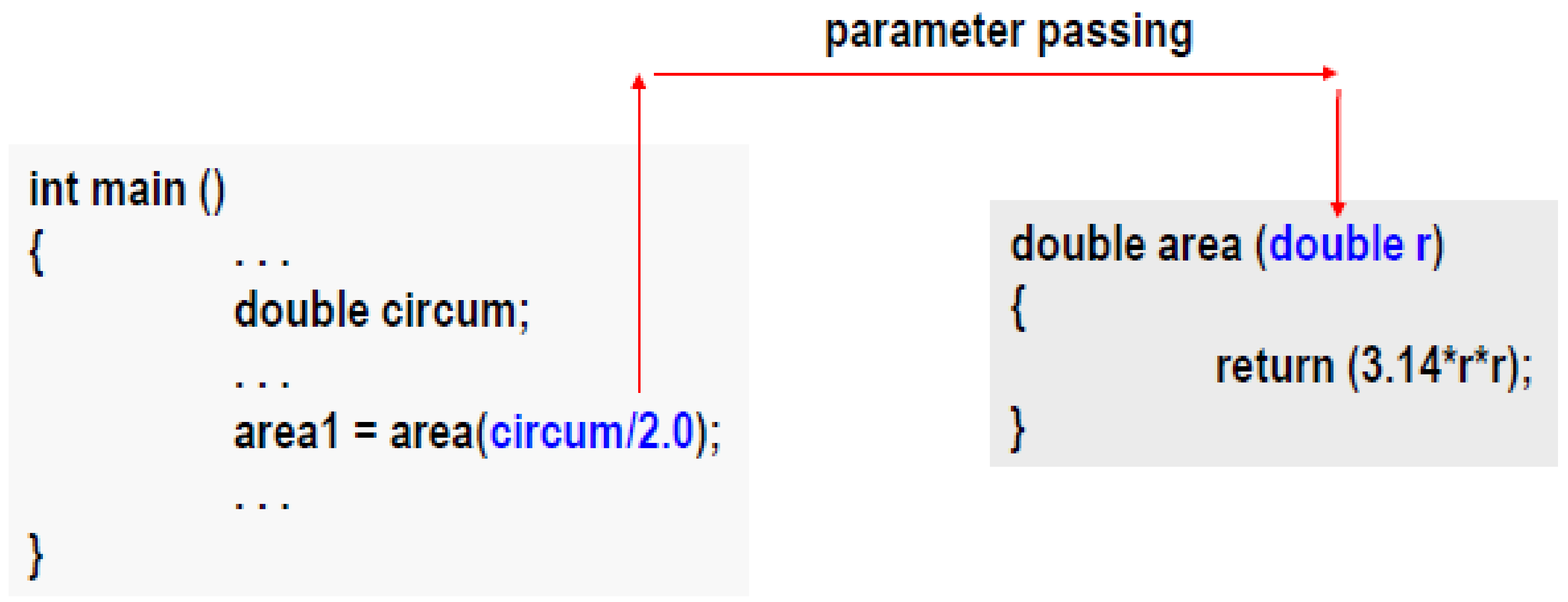


```
#include <stdio.h>
int sum_calc(int a, int b){
        int sum_cal = a+b;
        return sum_cal;
}
int main() {
        int x = 10, y=35, sum;
        sum = sum_calc(x,y);
        printf("Sum = %d", sum);
        return 0;
}
```

Parameter /Value Passing

1. The mechanism used to convey information to the function is the ‘argument’

When the function is executed, the value of the actual parameter is copied to the formal parameter



```
int main ()
{           . . .
            double circum;
            . . .
            area1 = area(circum/2.0);
            . . .
}
```

```
double area (double r)
{
            return (3.14*r*r);
}
```

Function Prototypes – Declaring function

Usually, a function is defined before it is called

- main() is the last function in the program written
- Easy for the compiler to identify function definitions in a single scan through the file

However, many programmers prefer a top-down approach, where the functions are written after main()

- Must be some way to tell the compiler
- Function prototypes are used for this purpose
    - Only needed if function definition comes after use

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including main())
- Prototypes can specify parameter names or just types (more common)
- Examples:

```
int  gcd (int , int );
void div7 (int number);
```

- Note the semicolon at the end of the line.
- The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition

Function Prototypes – Example

```
#include <stdio.h>
int sum( int, int );
int main( )
{
        int x, y;
        scanf("%d%d", &x, &y);
        printf("Sum = %d\n", sum(x, y));
}
int sum (int a, int b)
{
        return(a + b);
}
```

Scope Rule of Functions – why need to pass argumentss?

1. In this program is it necessary to pass the value of the variable i to the function display( )?
2. Will it not become automatically available to the function display( )?
3. No.
4. Because by default the scope of a variable is local to the function in which it is defined
5. The presence of i is known only to the function main( ) and not to any other function.
6. Similarly, the variable k is local to the function display( ) and hence it is not available to main( ).
7. That is why to make the value of i available to display( ) we have to explicitly pass it to display( ). Likewise, if we want k to be available to main( ) we will have to return it to main( ) using the return statement.

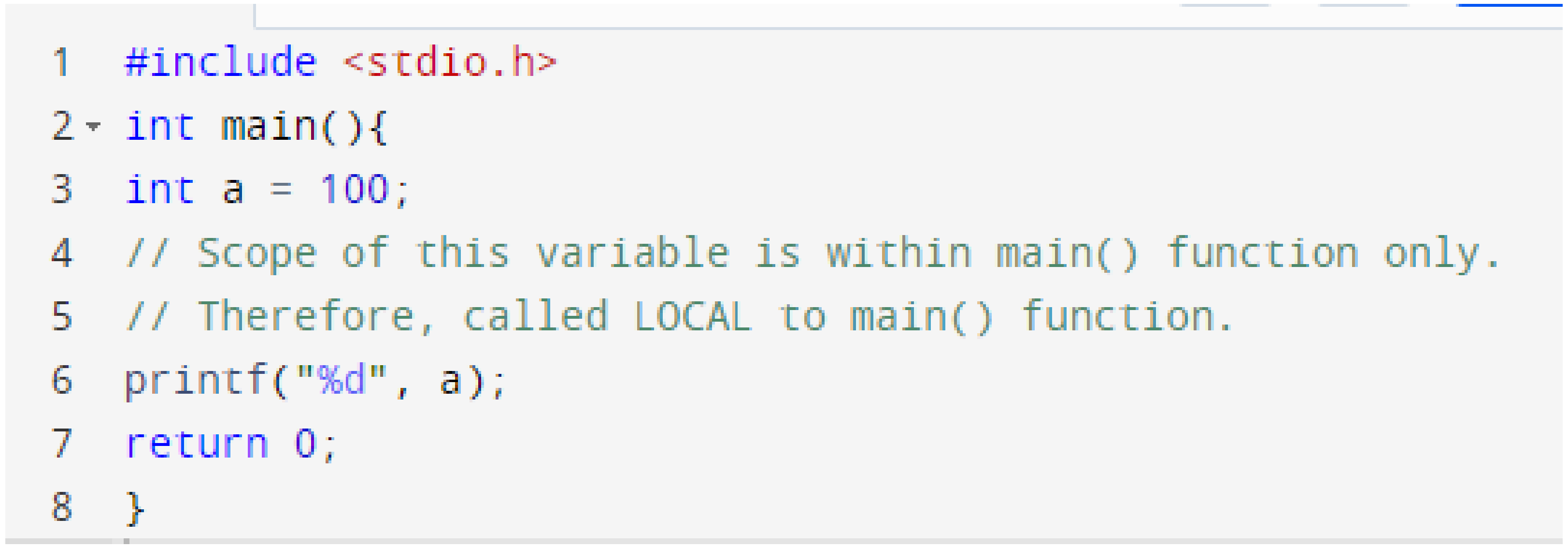
```
main( )
{
    int  i = 20 ;
    display ( i ) ;
}

display ( int  j )
{
    int  k = 35 ;
    printf ( "\n%d", j ) ;
    printf ( "\n%d", k ) ;
}
```

Scope Rule of Functions – why need to pass args?

1. Scope = Lifetime

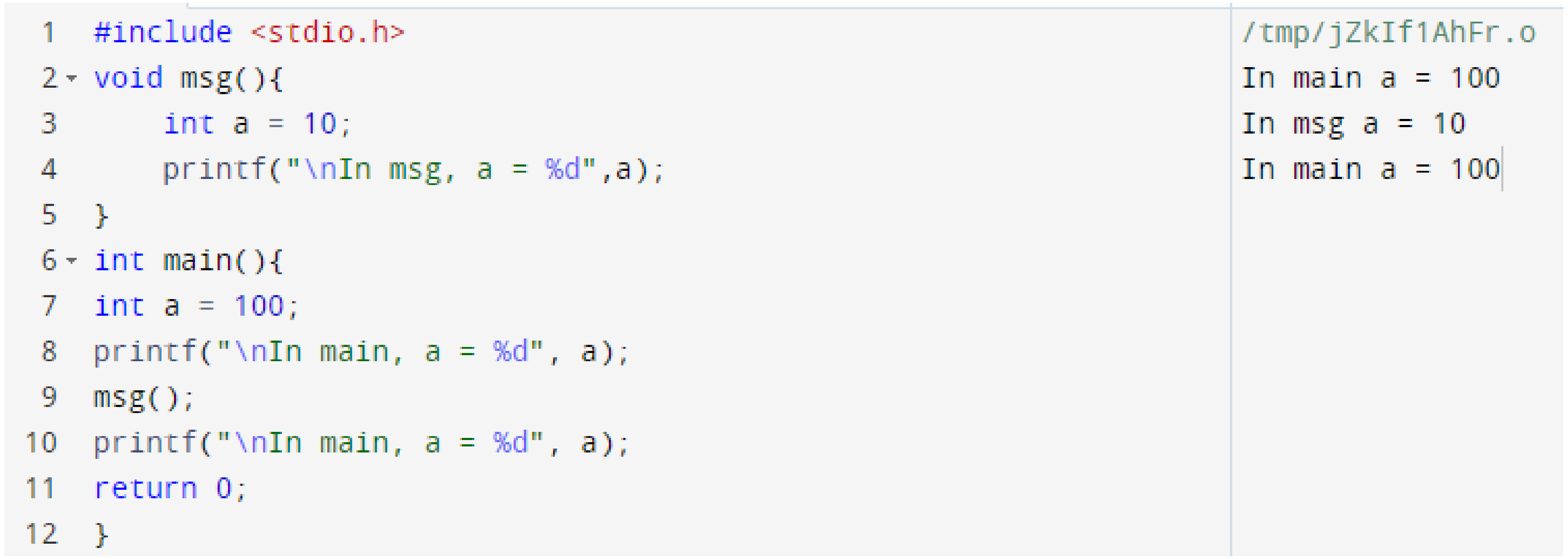
2. The area under which a variable is applicable.

3. Strict definition : A block or a region where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed.



```c
#include <stdio.h>
int main(){
int a = 100;
// Scope of this variable is within main() function only.
// Therefore, called LOCAL to main() function.
printf("%d", a);
return 0;
}
```

Scope Rule of Functions – why need to pass arguments Example

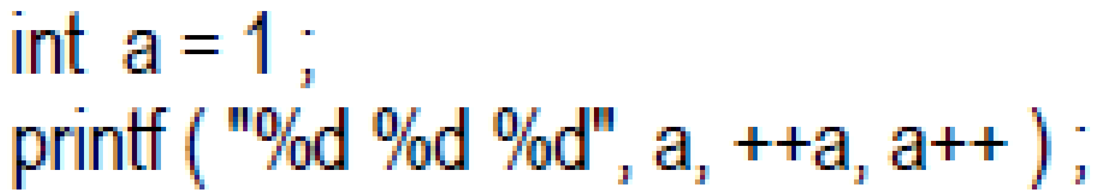


```c
#include <stdio.h>
void msg(){
    int a = 10;
    printf("\nIn msg, a = %d",a);
}
int main(){
int a = 100;
printf("\nIn main, a = %d", a);
msg();
printf("\nIn main, a = %d", a);
return 0;
}
```

```
/tmp/jZkIf1AhFr.o
In main a = 100
In msg a = 10
In main a = 100
```

Calling Convention in C

1. Calling convention indicates the order in which arguments are passed to a function when a function call is encountered
   1. Arguments might be passed from left to right.
   2. Arguments might be passed from right to left.
2. C language follows the second one.
3. **Output: 3 3 1**



```
int  a = 1 ;
printf ( "%d %d %d", a, ++a, a++ ) ;
```

Function Categories

1. So according to whether arguments are present or not and any values are returned or not, functions are of following types:
   a) No arguments no return values
   b) No arguments, but return value
   c) With arguments, no return value
   d) With arguments, with one/more return value

Function Calls - Details

1. Call by value
2. Call by reference
3. Before we learn about those, we need to know about - Pointers

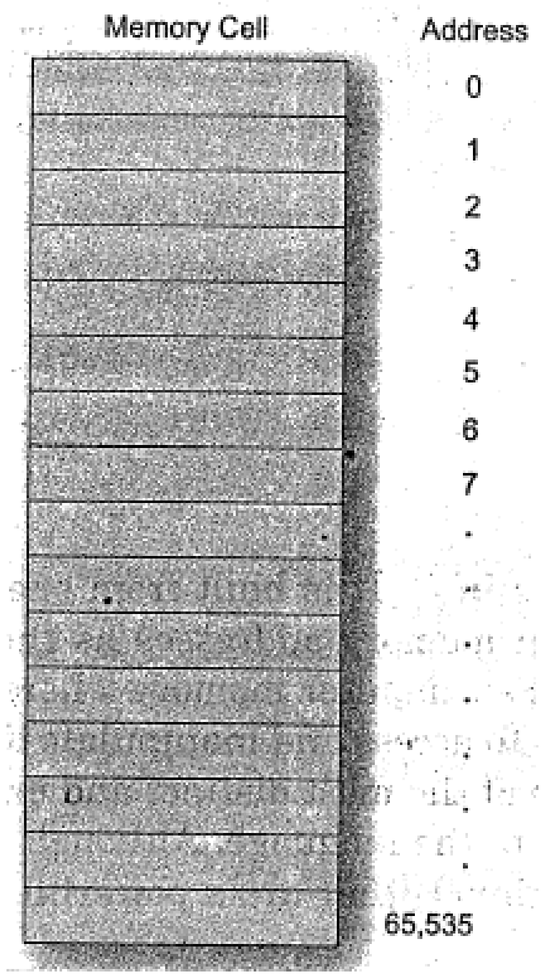
Pointers

Memory – Bits, bytes, how variables are stored

1. What is a bit?
    a) Binary digit
    b) Smallest unit of data a computer can process and store
    c) 1 byte = 8 bits
2. Generally
    a) 1 kilobyte (KB) = 1,024 bytes
    b) 1 MB = 1000 KB
    c) 1 GB = 1000 Mb
    d) 1 TB = 1000 GB and so on

Memory – Bits, bytes, how variables are stored

1. Computer's memory is a sequential collection of storage cells
2. Each cell is called a byte
3. A byte has an associated address
4. Addresses are numbered consecutively – start from 0
5. Last address depends on memory size



Memory – Bits, bytes, how variables are stored

1. Whenever we declare a variable, the system allocates some memory to hold that variables – this means, a memory location is allocated with unique address number.

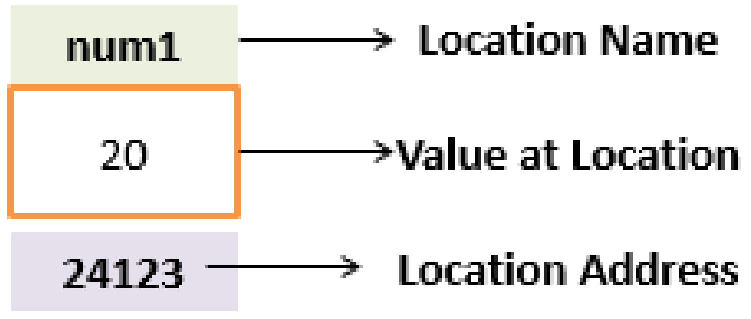
2. So, when I say: **int a = 10;** this means:

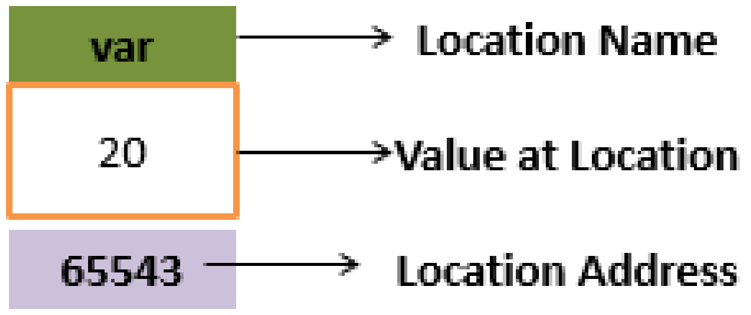
a) System find an appropriate location for a

b) Stores value of a in that location

3. A variable can occupy multiple bytes – its address is the address of the first byte.

4. Location name = variable name



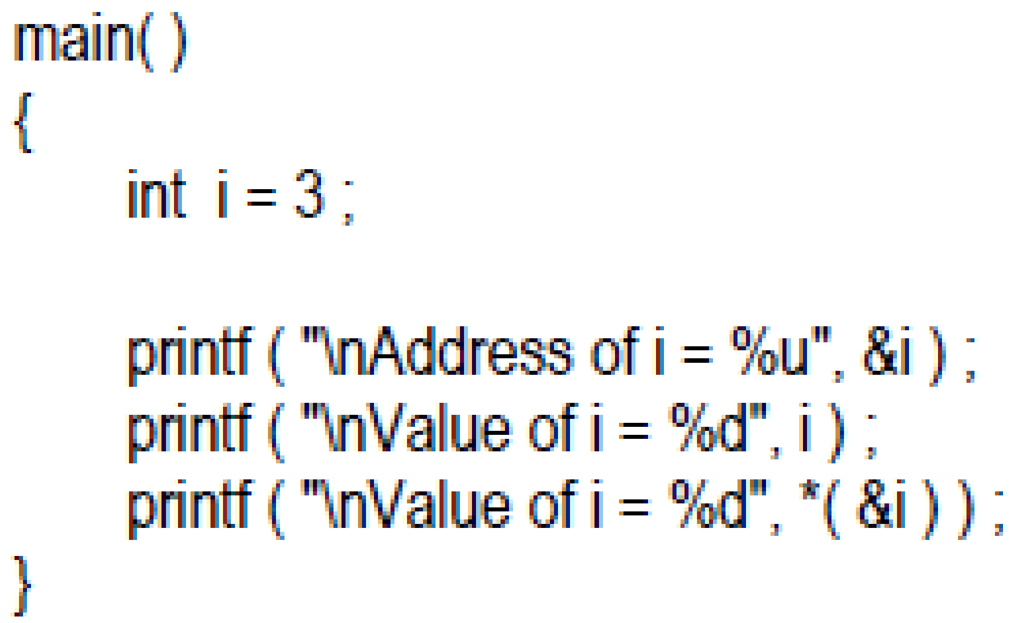


Memory – Bits, bytes, how variables are stored

1. System associates name a with address 5000
2. We can access the value 10, by either using the name a, or the address 5000
3. Since memory addresses are numbers, they can be assigned to other variables and stored in memory.
4. Such a variable that contains address of another variable is called a pointer variable

Pointers - Notations

- **& → Address of** operator
- *** → Value at address** operator



```
main( )
{
        int  i = 3 ;

        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

Pointers - Notations

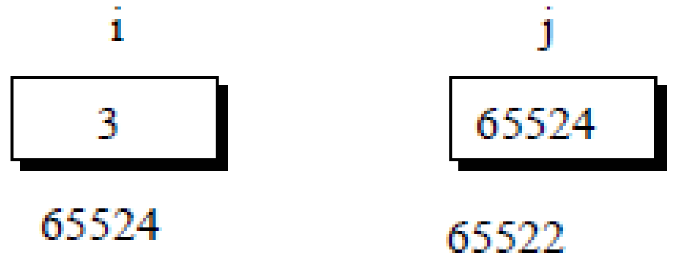
```
main( )
{
    int  i = 3 ;

    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

How to collect the address of variable “i” in another variable?

```
j = &i ;
```

What is this “j”? It contains the address of another variable.
How to declare it?

```
int  *j ;
```



This declaration tells the compiler that j will be used to store the address of an integer value.

j points to an integer.

Pointers – Basic Concept

1. A pointer is a variable that stores the memory address of another variable as its value.
2. A pointer variable points to a data type (like int) of the same type, and is created with the * operator.
3. The address of the variable you are working with is assigned to the pointer:

Pointer to pointer – Example code

```
main( )
{
    int  i = 3, *j, **k ;

    j = &i ;
    k = &j ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of i = %u", *k ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nAddress of j = %u", k ) ;
    printf ( "\nAddress of k = %u", &k ) ;
    printf ( "\nValue of j  = %u", j ) ;
    printf ( "\nValue of k  = %u", k ) ;
    printf ( "\nValue of i  = %d", i ) ;
    printf ( "\nValue of i  = %d", * ( &i ) ) ;
    printf ( "\nValue of i  = %d", *j ) ;
    printf ( "\nValue of i  = %d", **k ) ;
}
```

```
Address of i = 65524
```

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j  = 65524
Value of k  = 65522
```

```
Value of i  = 3
Value of i  = 3
Value of i  = 3
Value of i  = 3
```

Call by Value

Call by Value

1. Till now we have seen call by value.
2. So, what happens in call by value?
3. In the first method the ‘value’ of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.
4. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function

Call by Value

```
main( )
{
        int  a = 10, b = 20 ;

        swapv ( a, b ) ;
        printf ( "\na = %d b = %d", a, b ) ;
}

swapv ( int  x, int  y )
{
        int  t ;

        t = x ;
        x = y ;
        y = t ;

        printf ( "\nx = %d y = %d", x, y ) ;
}
```

```
x = 20 y = 10
a = 10 b = 20
```

1. Note that values of a and b remain unchanged even after exchanging the values of x and y.

Call by Reference

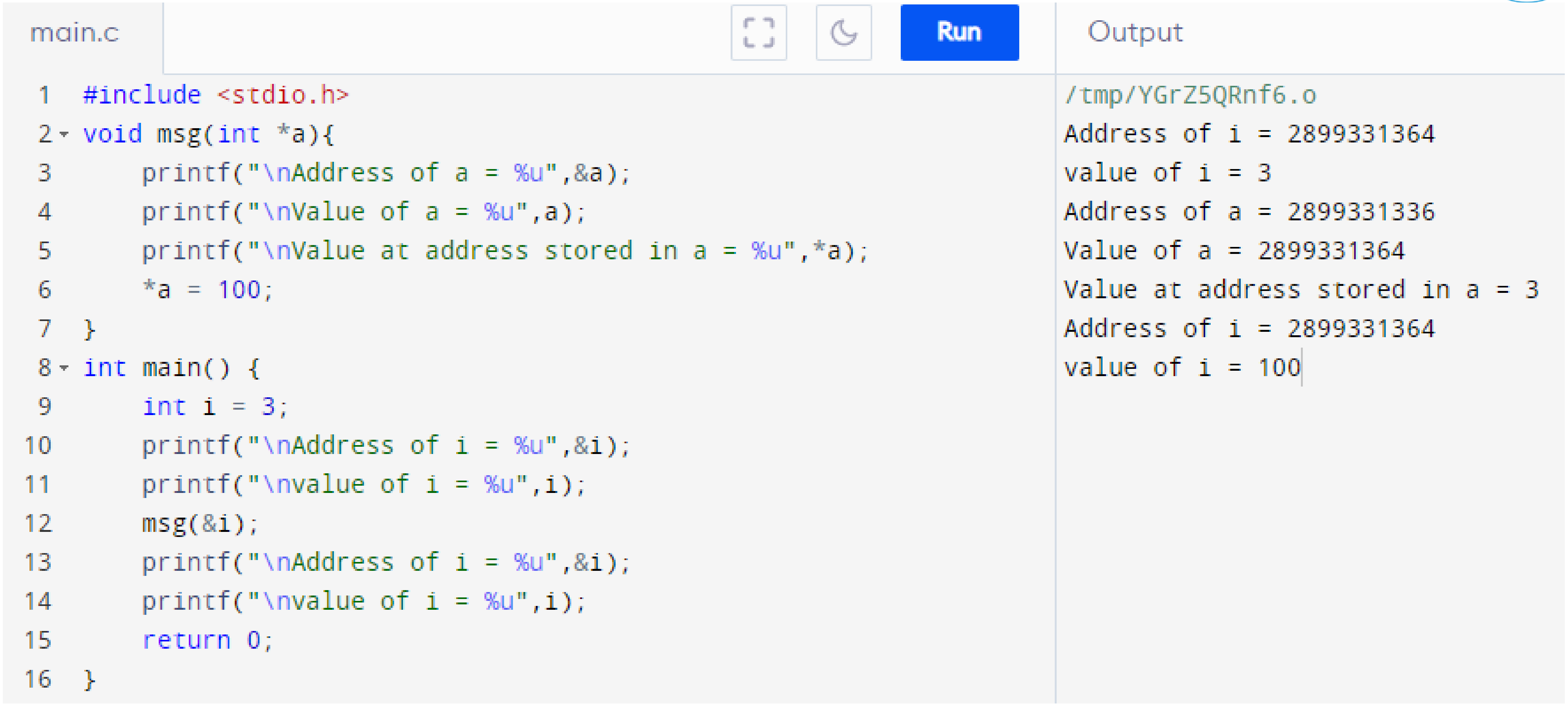
Call by Reference

1. Now to see call by reference. What happens here?
   a) We have learnt that variables are stored somewhere in memory.
   b) So instead of passing the value of a variable, we can pass the location number (also called address) of the variable to a function. If we were able to do so it would become a 'call by reference'.
2. We need pointers.

Call by Reference

1. In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function.
2. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them

Call by Reference - Example



main.c

```c
#include <stdio.h>
void msg(int *a){
    printf("\nAddress of a = %u",&a);
    printf("\nValue of a = %u",a);
    printf("\nValue at address stored in a = %u",*a);
    *a = 100;
}
int main() {
    int i = 3;
    printf("\nAddress of i = %u",&i);
    printf("\nvalue of i = %u",i);
    msg(&i);
    printf("\nAddress of i = %u",&i);
    printf("\nvalue of i = %u",i);
    return 0;
}
```

Output

```
/tmp/YGrZ5QRnf6.o
Address of i = 2899331364
value of i = 3
Address of a = 2899331336
Value of a = 2899331364
Value at address stored in a = 3
Address of i = 2899331364
value of i = 100
```

Call by Reference

```
main( )
{
	int  a = 10, b = 20 ;

	swapr ( &a, &b ) ;
	printf ( "\na = %d b = %d", a, b ) ;
}

swapr( int  *x, int  *y )
{
	int  t ;

	t = *x ;
	*x = *y ;
	*y = t ;
}
```

The output of the above program would be:

```
a = 20 b = 10
```

1. Note that this program manages to exchange the values of a and b using their addresses stored in x and y.

Return more than 1 value from function - Call by Reference

```
main( )
{
        int  radius ;
        float  area, perimeter ;

        printf ( "\nEnter radius of a circle " ) ;
        scanf ( "%d", &radius ) ;
        areaperi ( radius, &area, &perimeter ) ;

        printf ( "Area = %f", area ) ;
        printf ( "\nPerimeter = %f", perimeter ) ;
}

areaperi ( int  r, float  *a, float  *p )
{
        *a = 3.14 * r * r ;
        *p = 2 * 3.14 * r ;
}
```

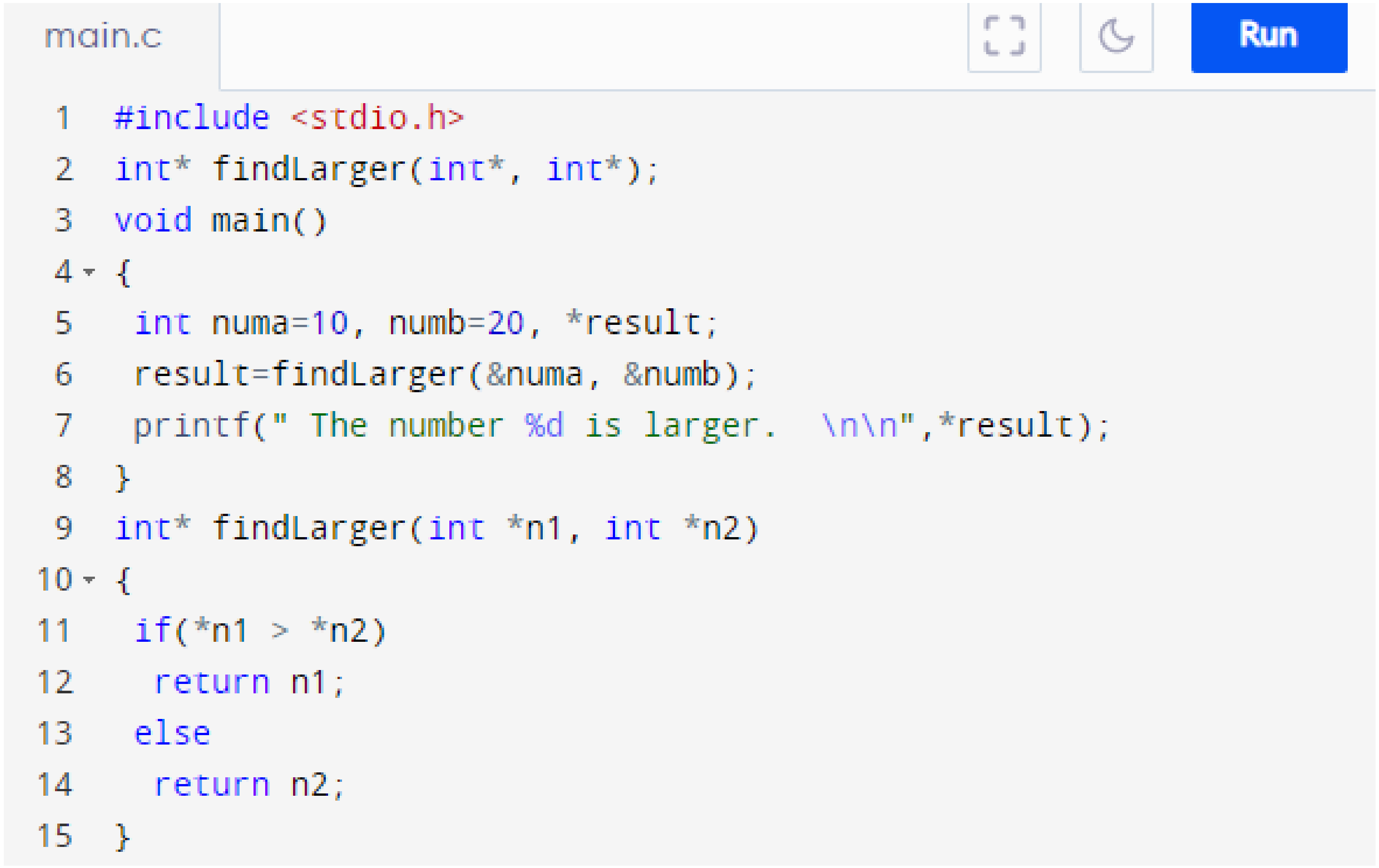
What is the output?

What is happening here?

Mixed call –

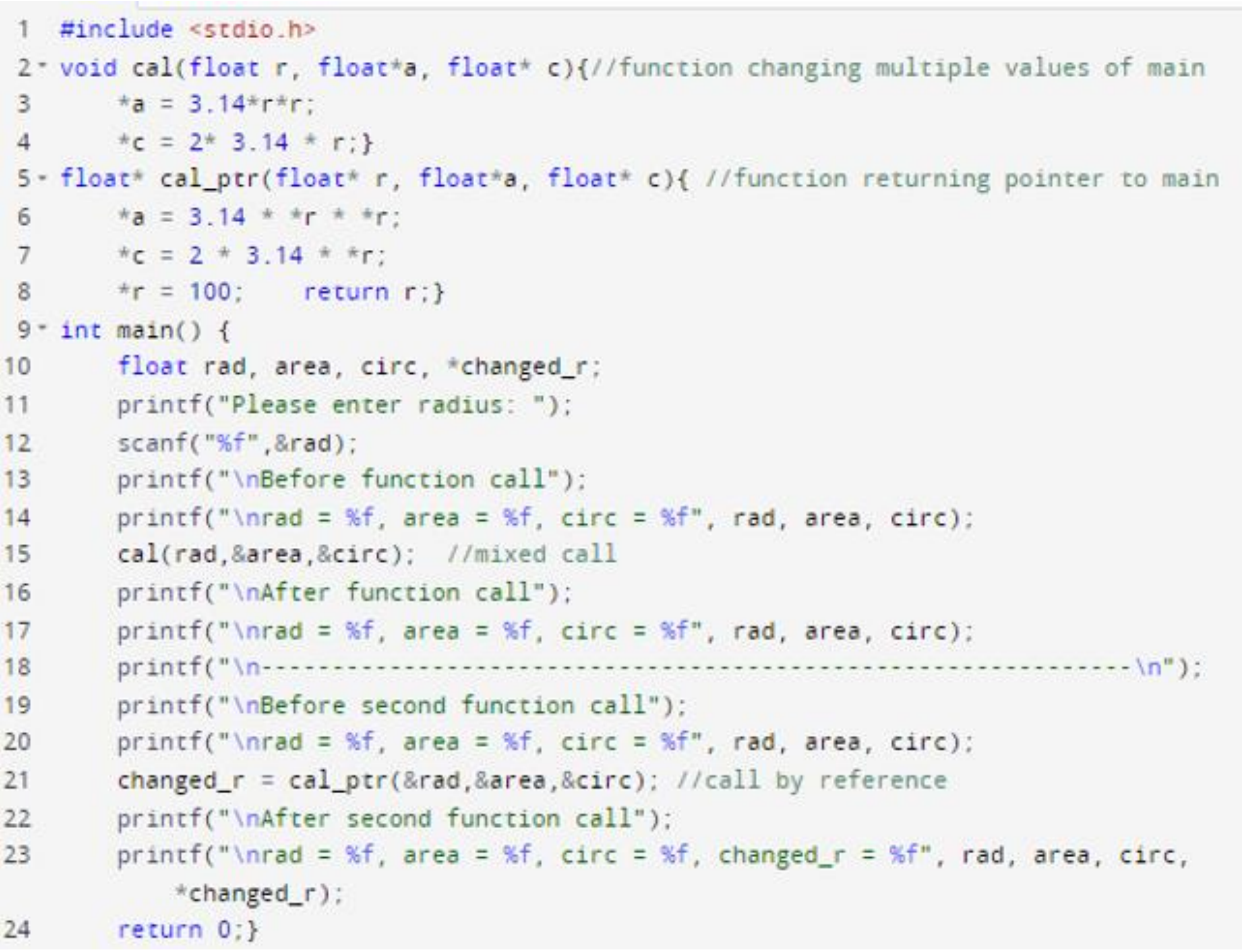
1. Pass the value of radius
2. Pass the addresses of area and perimeter
3. **overcome the limitation of the return statement**

Functions returning pointers



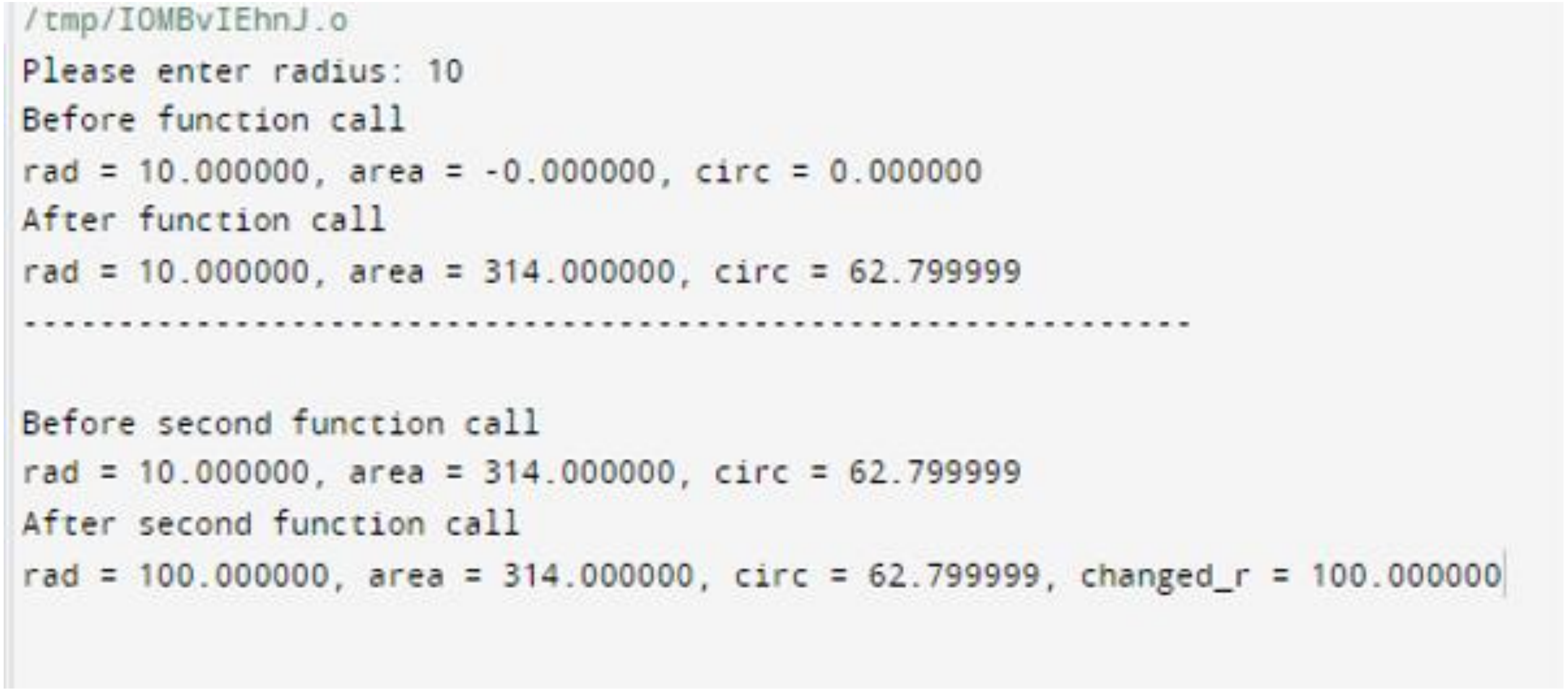
```c
#include <stdio.h>
int* findLarger(int*, int*);
void main()
{
 int numa=10, numb=20, *result;
 result=findLarger(&numa, &numb);
 printf(" The number %d is larger.  \n\n",*result);
}
int* findLarger(int *n1, int *n2)
{
 if(*n1 > *n2)
  return n1;
 else
  return n2;
}
```

Functions –Extra Example



```c
#include <stdio.h>
void cal(float r, float*a, float* c){//function changing multiple values of main
    *a = 3.14*r*r;
    *c = 2* 3.14 * r;}
float* cal_ptr(float* r, float*a, float* c){ //function returning pointer to main
    *a = 3.14 * *r * *r;
    *c = 2 * 3.14 * *r;
    *r = 100;   return r;}
int main() {
    float rad, area, circ, *changed_r;
    printf("Please enter radius: ");
    scanf("%f",&rad);
    printf("\nBefore function call");
    printf("\nrad = %f, area = %f, circ = %f", rad, area, circ);
    cal(rad,&area,&circ);  //mixed call
    printf("\nAfter function call");
    printf("\nrad = %f, area = %f, circ = %f", rad, area, circ);
    printf("\n------------------------------------------------------------\n");
    printf("\nBefore second function call");
    printf("\nrad = %f, area = %f, circ = %f", rad, area, circ);
    changed_r = cal_ptr(&rad,&area,&circ); //call by reference
    printf("\nAfter second function call");
    printf("\nrad = %f, area = %f, circ = %f, changed_r = %f", rad, area, circ,
        *changed_r);
    return 0;}
```

Functions –Extra Example - Output



```
/tmp/IOMBvIEhnJ.o
Please enter radius: 10
Before function call
rad = 10.000000, area = -0.000000, circ = 0.000000
After function call
rad = 10.000000, area = 314.000000, circ = 62.799999
-------------------------------------------------------------

Before second function call
rad = 10.000000, area = 314.000000, circ = 62.799999
After second function call
rad = 100.000000, area = 314.000000, circ = 62.799999, changed_r = 100.000000
```

Recursion

Recursion

1. A process by which a function calls itself repeatedly.

- **Used for repetitive computations in which each action is stated in terms of a previous result.**

```
fact(n) = n * fact (n-1)
```

Recursion - Recap

- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form.
  - The problem statement must include a stopping condition

```
fact(n)  =  1,                 if  n = 0
         =  n * fact(n-1),     if  n > 0
```

Recursion - Recap

Some example problems which can be solved using recursion:

1. Factorial

2. GCD

3. Fibonacci Series (0, 1, 1, 2, 3, 5, 8, 13, ….)

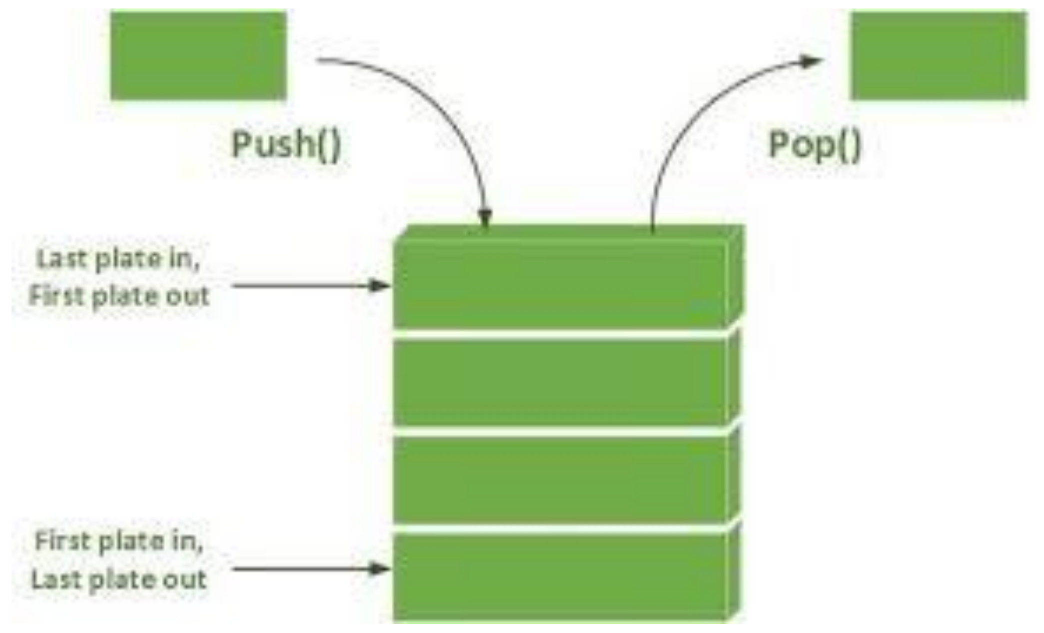
Recursion – Mechanism of Execution

```
long  int  fact (n)
int  n;
{
    if   (n == 0)
        return (1);
    else
        return  (n * fact(n-1));
}
```

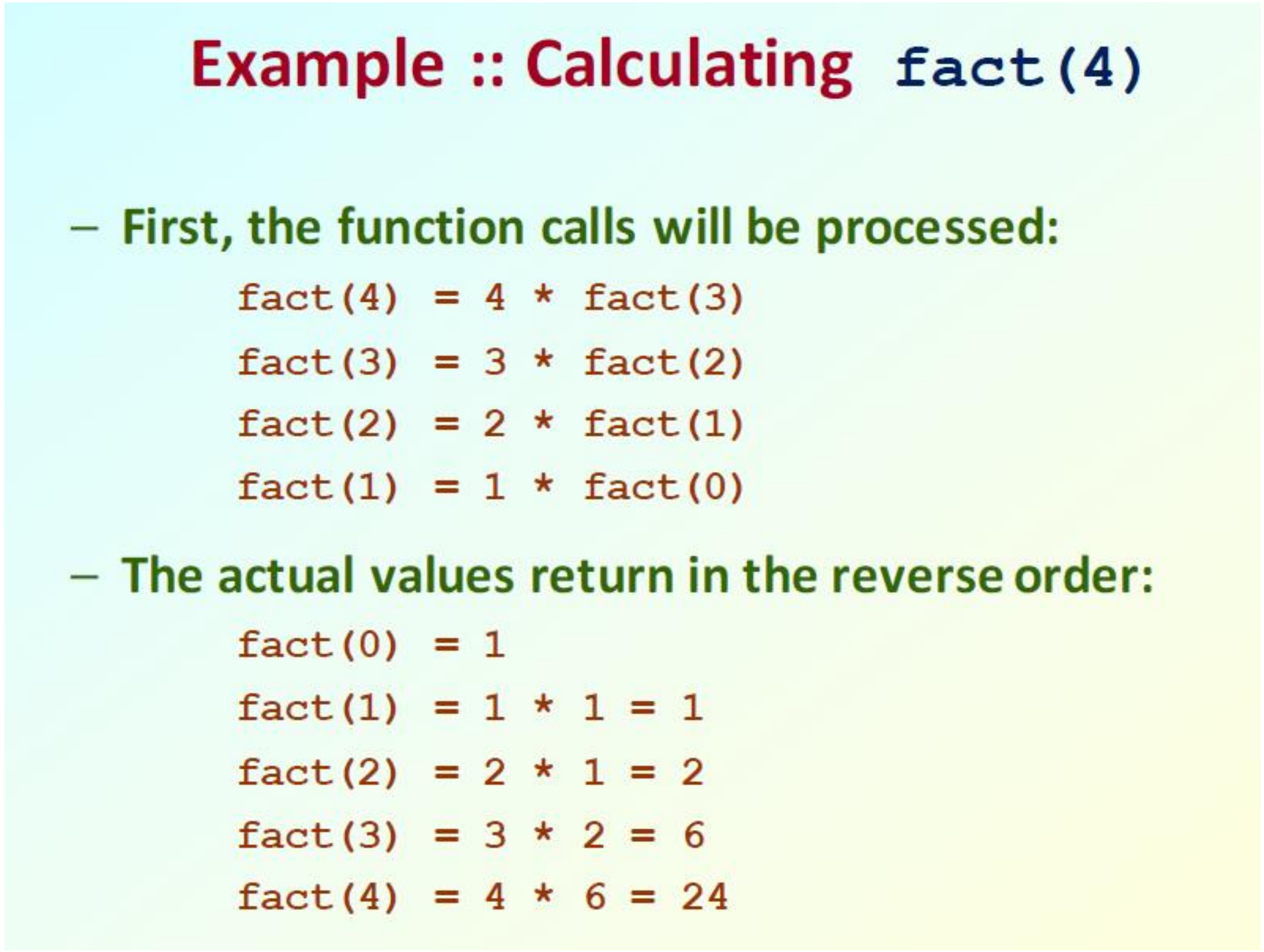
- **Mechanism of execution**
  - **When a recursive program is executed, the recursive function calls are not executed immediately.**
    - They are kept aside (on a stack) until the stopping condition is encountered.
    - The function calls are then executed in reverse order.

Recursion – What is a stack?

1. **A stack is a Last In First Out (LIFO) data structure.**
2. **The last item to get stored on the stack (Push operation) is the first one to get out of it (Pop operation).**



Recursion - Mechanism of Execution



Example :: Calculating `fact(4)`

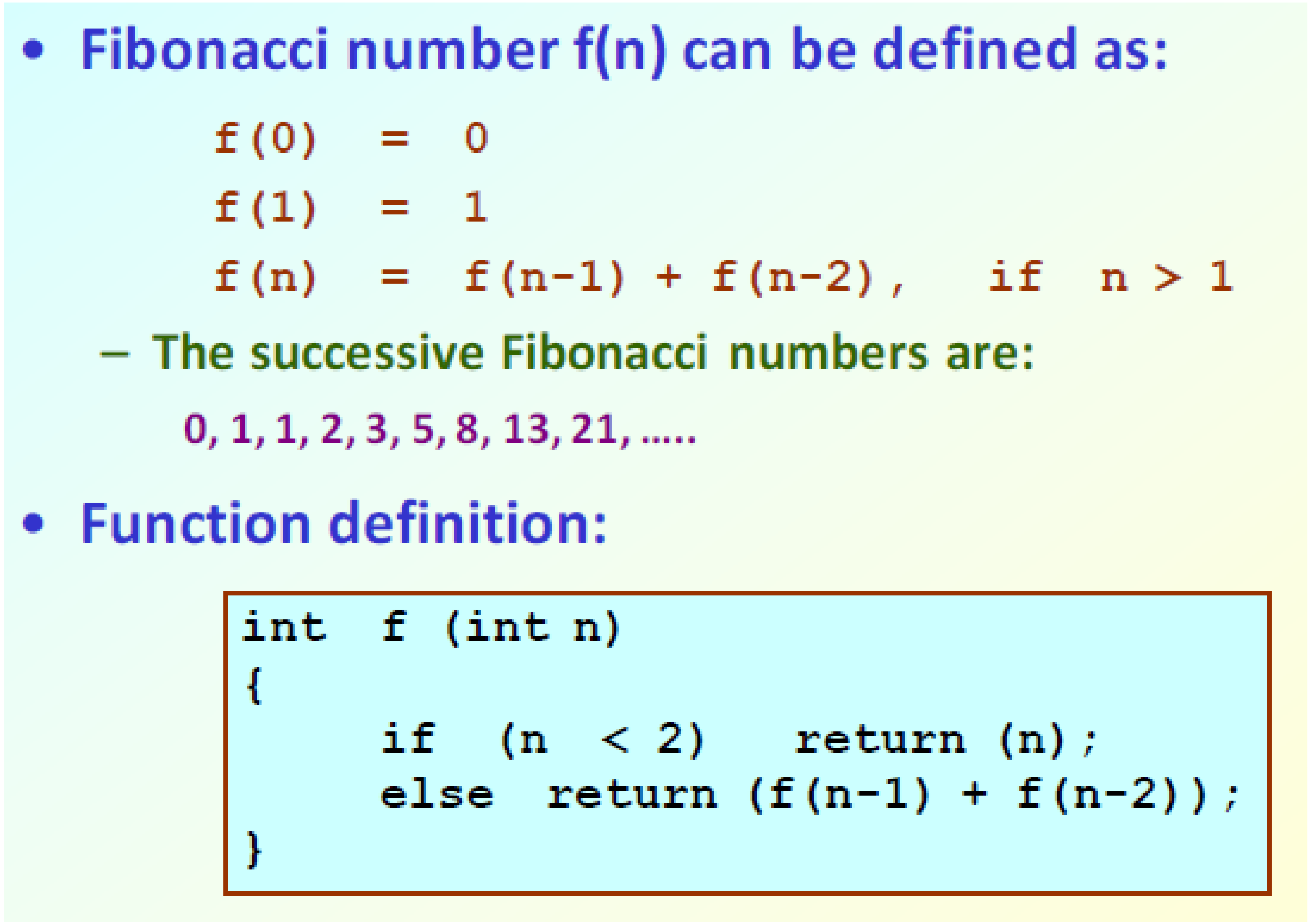
– First, the function calls will be processed:

```
fact(4)  = 4 * fact(3)
fact(3)  = 3 * fact(2)
fact(2)  = 2 * fact(1)
fact(1)  = 1 * fact(0)
```

– The actual values return in the reverse order:

```
fact(0)  = 1
fact(1)  = 1 * 1 = 1
fact(2)  = 2 * 1 = 2
fact(3)  = 3 * 2 = 6
fact(4)  = 4 * 6 = 24
```

Recursion – Fibonacci Numbers



- Fibonacci number f(n) can be defined as:

```
f(0)  =  0
f(1)  =  1
f(n)  =  f(n-1) + f(n-2),   if  n > 1
```

  - The successive Fibonacci numbers are:

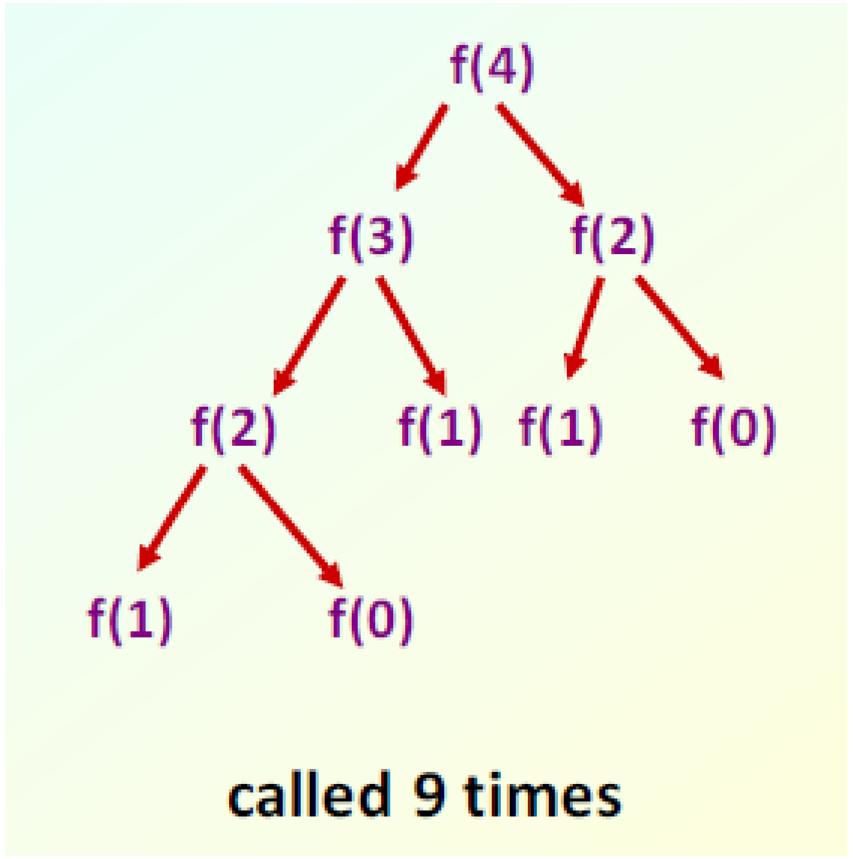
    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

- Function definition:

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

Recursion – Fibonacci Numbers

How many times the function is called when evaluating f(4) ?



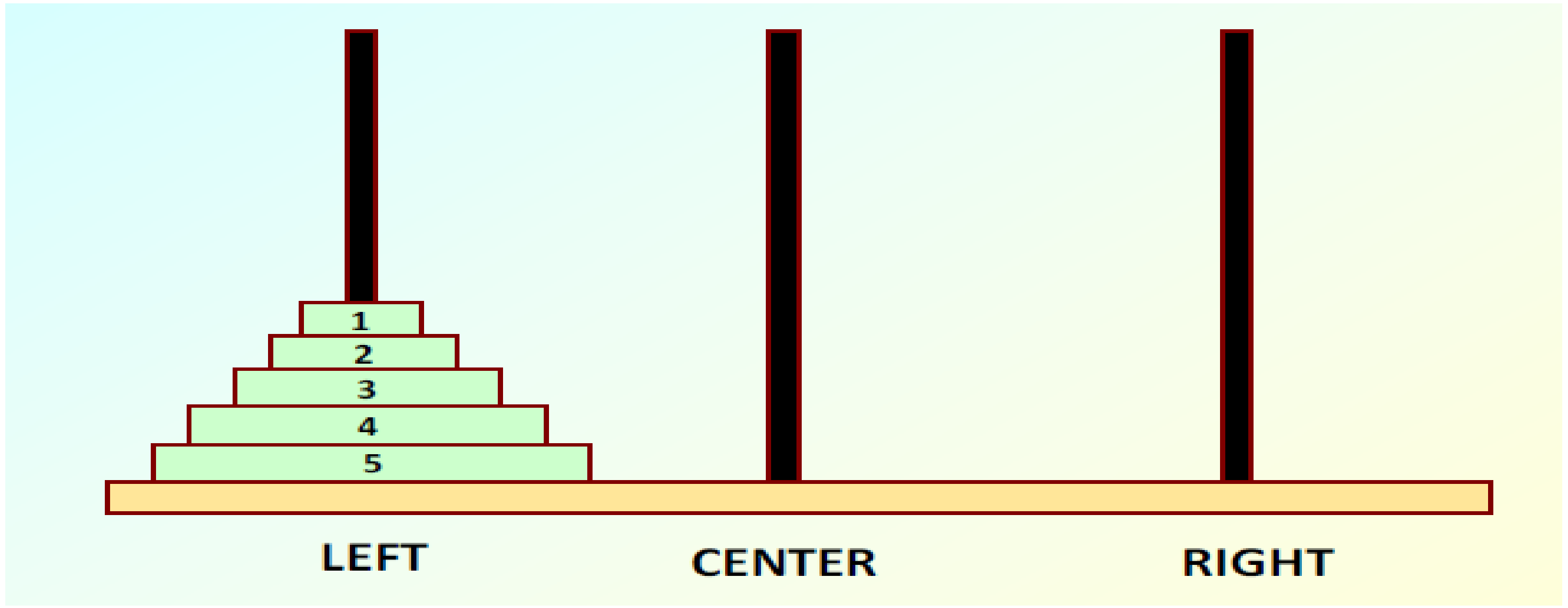
Same thing is calculate multiple times – inefficient.

Recursion – What to avoid

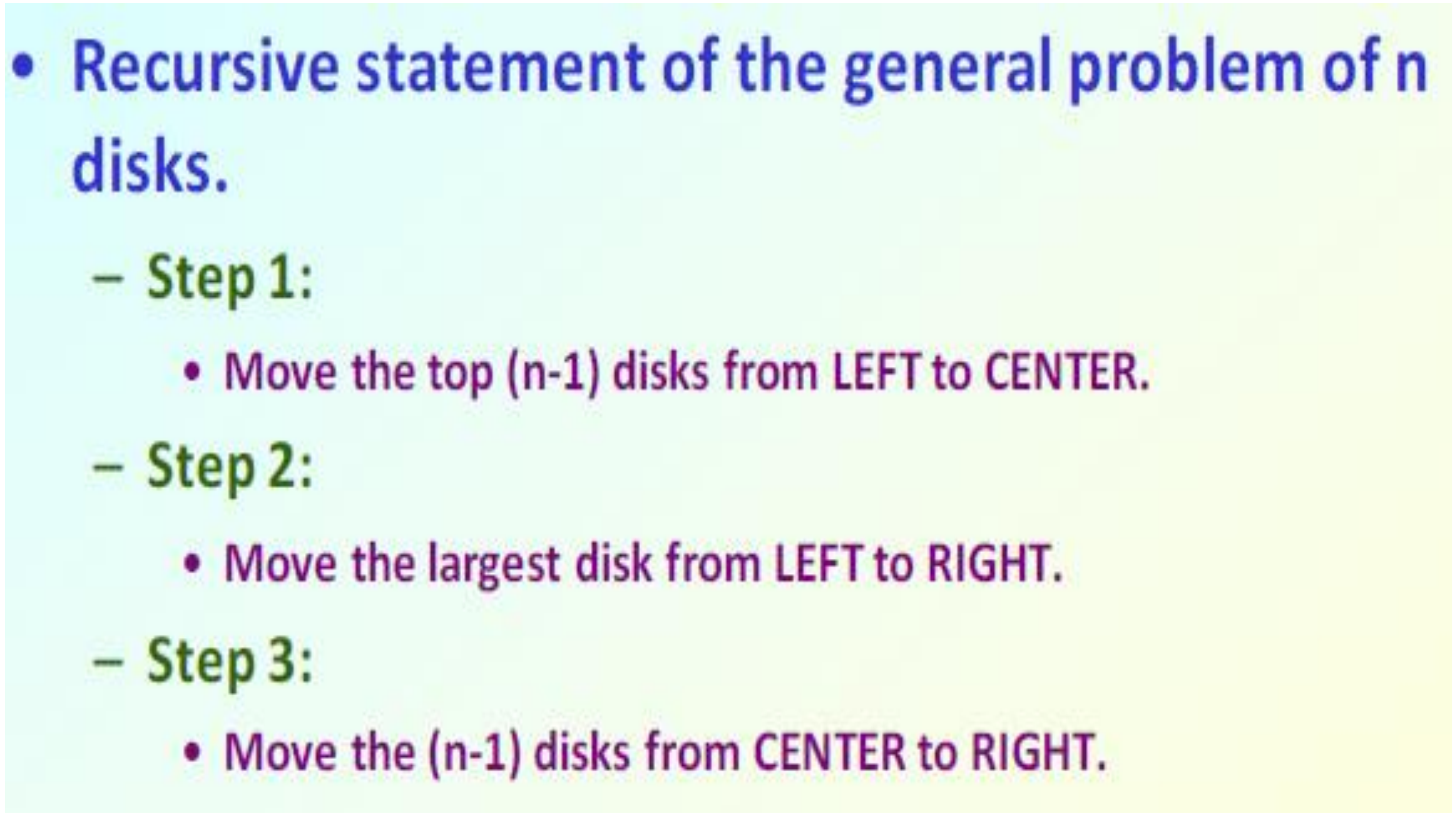
1. **Avoid Fibonacci-style recursive programs which result in an exponential “explosion” of calls.**
2. **Avoid using recursion in performance situations.**
3. **Recursive calls take time and consume additional memory.**

Recursion – Tower of Hanoi

- The problem statement:
  - Initially all the disks are stacked on the LEFT pole.
  - Required to transfer all the disks to the RIGHT pole.
    - Only one disk can be moved at a time.
    - A larger disk cannot be placed on a smaller disk.
  - CENTER pole is used for temporary storage of disks.

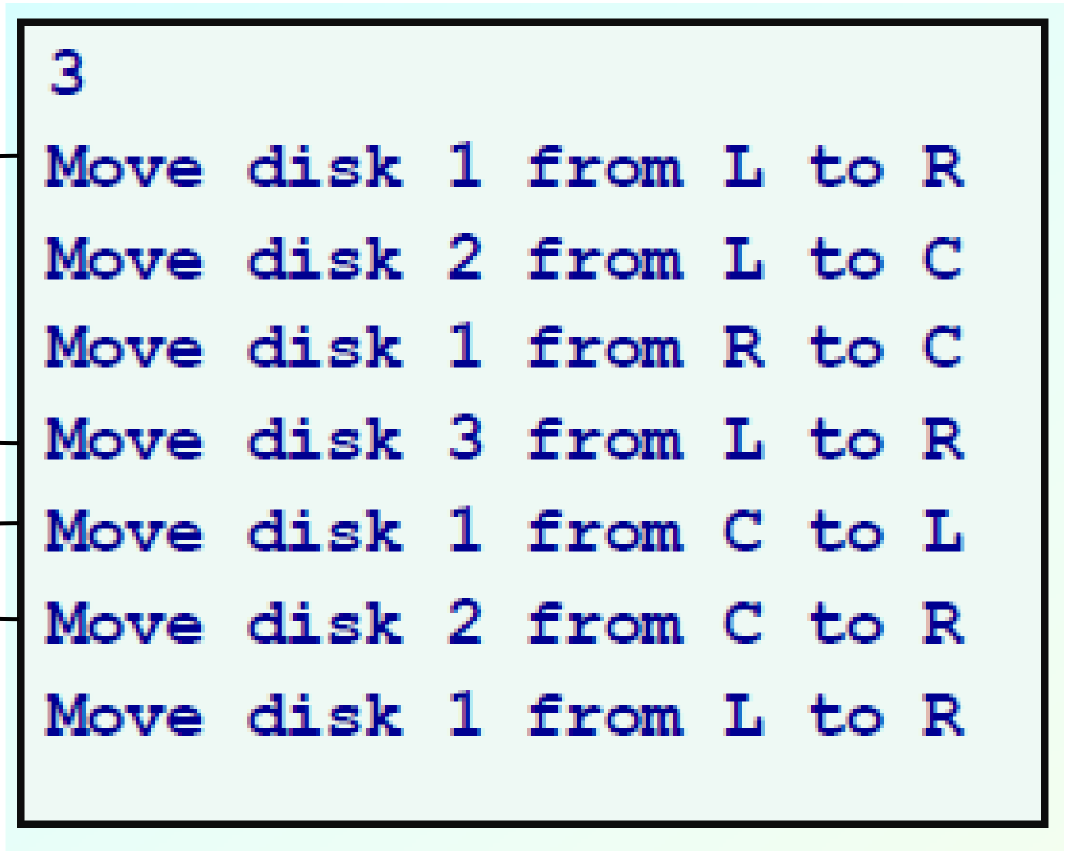


Recursion – Tower of Hanoi



- Recursive statement of the general problem of n disks.
  - Step 1:
    - Move the top (n-1) disks from LEFT to CENTER.
  - Step 2:
    - Move the largest disk from LEFT to RIGHT.
  - Step 3:
    - Move the (n-1) disks from CENTER to RIGHT.

Recursion – Tower of Hanoi



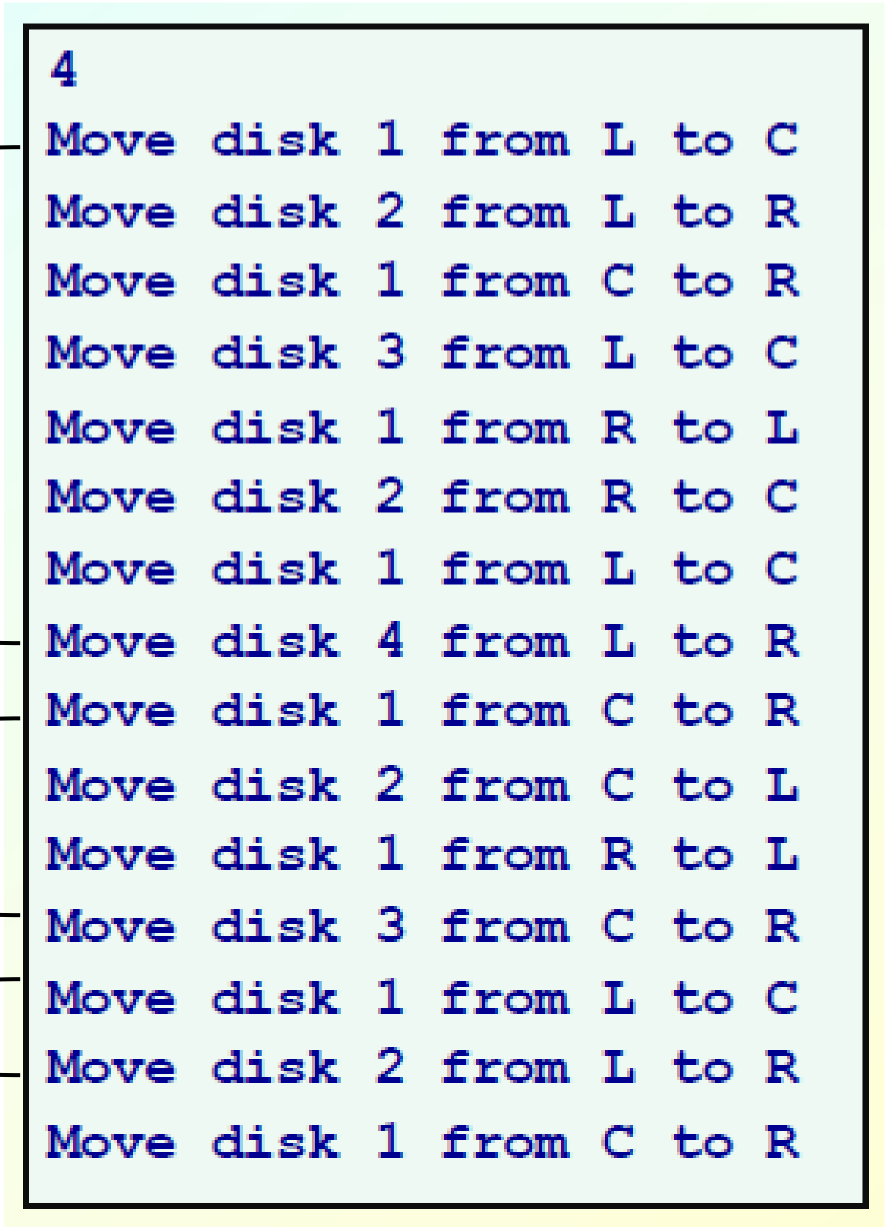
```
3
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

Moved disk 3 from L to R

Moved disk 2 from L to R

Moved disk 1 from L to R

Recursion – Tower of Hanoi



```
4
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
Move disk 3 from L to C
Move disk 1 from R to L
Move disk 2 from R to C
Move disk 1 from L to C
Move disk 4 from L to R
Move disk 1 from C to R
Move disk 2 from C to L
Move disk 1 from R to L
Move disk 3 from C to R
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
```

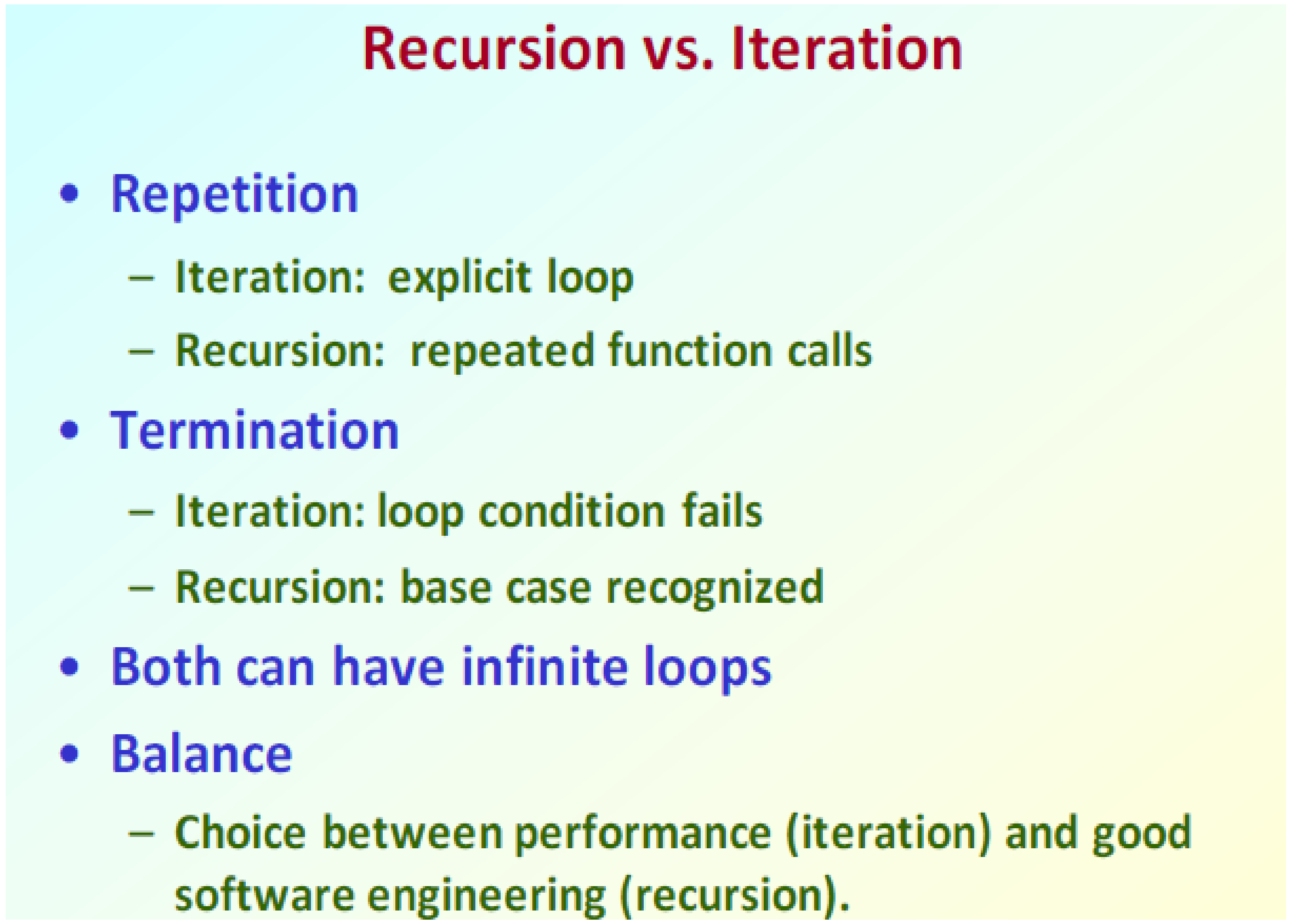
Moved disk 4 from L to R

Moved disk 3 from L to R

Moved disk 2 from L to R

Moved disk 1 from L to R

Recursion vs Iteration



Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion).