BLG 102E Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 6

ISTANBUL TECHNICAL UNIVERSITY

Functions – Part II

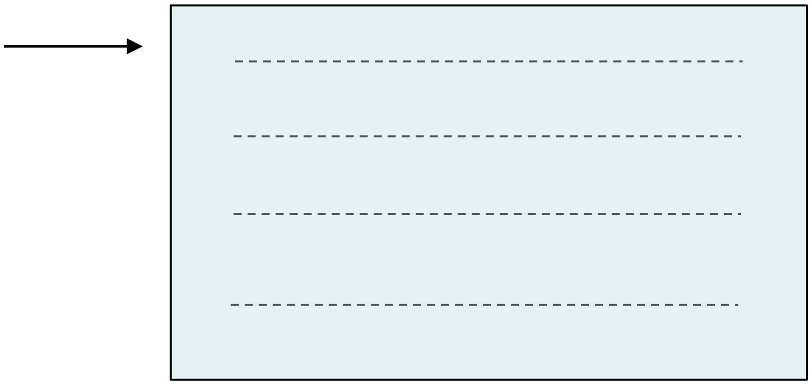
Slides Credit: Cay Horstmann

What Is a Function? Why Functions?

A function is a sequence of instructions with a name.

A function packages a computation into a form that can be easily understood and reused.

do_something()



Implementing Functions

Syntax 5.1 Function Definition

Type of return value

Name of function

Type of parameter variable

Name of parameter variable

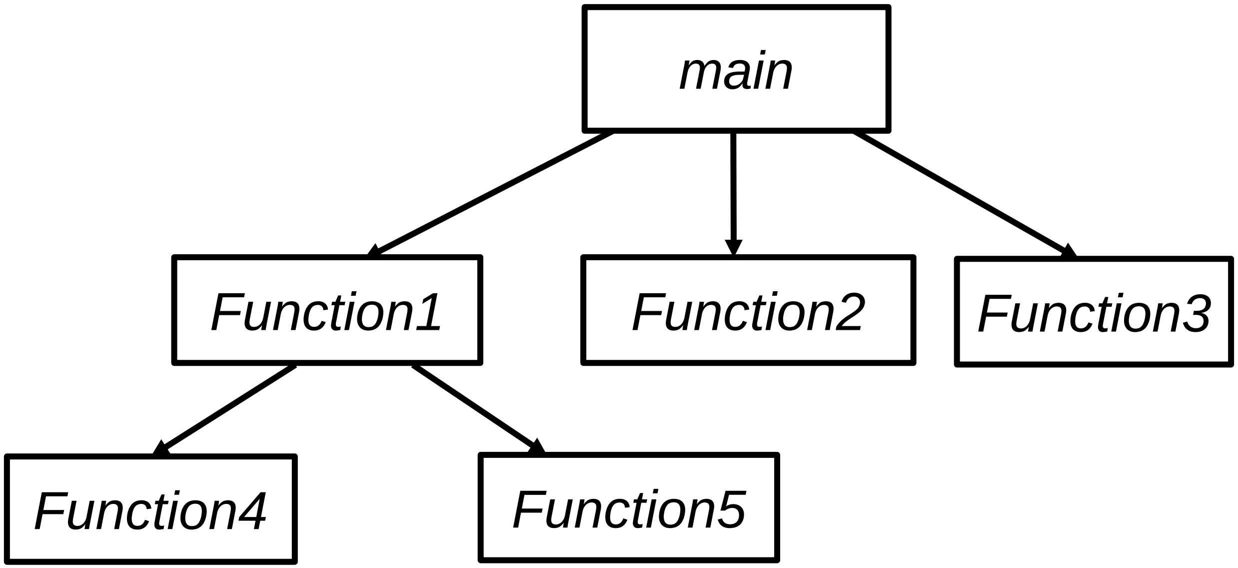
```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Function body, executed when function is called.

return statement exits function and returns result.

Calling Functions

Consider the order of activities when a function is called.



Variable Scope

?

Variable Scope

Which main st?

Variable Scope

You can only have *one* **`main`** function
but you can have as many variables and parameters
spread amongst as many functions as you need.

Can you have the same name in different functions?

Variable Scope

The **`railway_avenue`** *and* **`main_street`** *variables in the* **`oklahoma_city`** *function*

The **`south_street`** ***and*** **`main_street`** *variables in the* **`panama_city`** *function*

The **`n_putnam_street`** *and* **`main_street`** *variables in the* **`new_york_city`** *function*

Variable Scope

A variable or parameter that is defined within a function is visible from the point at which it is defined until the end of the block named by the function.

This area is called the *scope* of the variable.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Because scopes do not overlap, a name in one scope cannot conflict with any name in another scope.

Variable Scope

The scope of a variable is the part of the program in which it is *visible*.

Because scopes do not overlap, a name in one scope cannot conflict with any name in another scope.

A name in one scope is “invisible” in another scope

Variable Scope

```
double cube_volume(double side_len)
{
   double volume = side_len * side_len * side_len;
   return volume;
}
int main()
{
   double volume = cube_volume(2);
   printf("%f\n", volume);
   return 0;
}
```

Each **`volume`** variable is defined in a separate function, so there is not a problem with this code.

Variable Scope

Because of scope, when you are writing a function you can focus on choosing variable and parameter names that make sense for your function.

You do not have to worry that your names will be used elsewhere.

Variable Scope

Names inside a block are called *local* to that block.

A function names a block.

Recall that variables and parameters do not exist after the function is over—because they are local to that block.

But there are other blocks.

Variable Scope

It is *not legal* to define two variables or parameters with the same name in the same scope.

For example, the following is not legal:

ERRORS!!!

```
int test(double volume)
{
   double volume = cube_volume(2);
   double volume = cube_volume(10);
// ERROR: cannot define another volume variable
// ERROR: or parameter in the same scope
...
}
```

Variable Scope – Nested Blocks

However, you can define another variable with the same name in a *nested block*.

```
double withdraw(double balance, double amount)
{
   if (...)
   {
      double amount = 10;
      ...
   }
   ...
}
```

a variable named **`amount`** local to the **`if`**'s block

– *and* a parameter variable named **`amount`**.

Variable Scope – Nested Blocks

The scope of the parameter variable **`amount`** is the entire function, *except* the nested block.

Inside the nested block, **`amount`** refers to the local variable that was defined in that block.

You should avoid this *potentially confusing situation* in the functions that you write, simply by renaming one of the variables.

Why should there be a variable with the same name in the same function?

Global Variables

- Generally, global variables are ***not*** a good idea.

  But …

  here’s what they are and how to use them

  (if you must).

Global Variables

Global variables are defined *outside* any block.

They are visible to every function defined after them.

Global Variables

But in a banking program, how many functions should have direct access to a balance variable?

Global Variables

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
    if (balance >= amount)
    {
        balance = balance - amount;
    }
}

int main()
{
    withdraw(1000);
    printf("%d", balance);
    return 0;
}
```

Global Variables

In the previous program there is only one function that updates the **`balance`** variable.

But there could be many, many, many functions that might need to update **`balance`** each written by any one of a huge number of programmers in a large company.

Then we would have a problem.

Global Variables

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
    if (balance >= amount)
    {
        balance = balance - amount;
    }
}
void deposit(double amount)
{
    balance = balance + amount;
}
int main()
{
    withdraw(1000);
    deposit(200);
    printf("%d", balance);
    return 0;
}
```

Global Variables

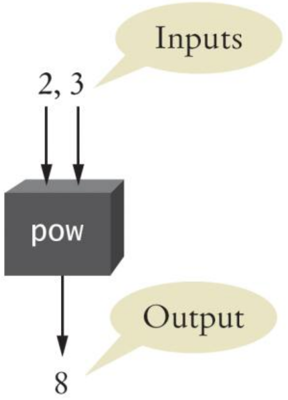
When multiple functions update global variables, the result can be *difficult* to predict.

Particularly in larger programs that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand.

Global Variables – Breaking Open the Black Box

When functions modify global variables, it becomes more difficult to understand the effect of function calls.

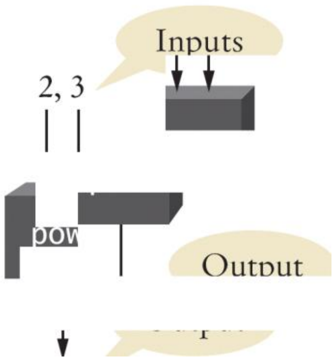
Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives parameter values and returns a result or does something.



Global Variables – Breaking Open the Black Box

When functions modify global variables, it becomes more difficult to understand the effect of function calls.

Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives parameter values and returns a result or does something.



And what good is a broken black box?

Global Variables – Just Say “No”

You should *avoid* global variables in your programs!

Parameter Modifications are Local

Consider a function that simulates withdrawing a given amount of money from a bank account, provided that sufficient funds are available.

If the amount of money is insufficient, a $10 penalty is deducted instead.

The function would be used as follows:

```
double harrys_account = 1000;
withdraw(harrys_account, 100);
     // Now harrys_account is 900
withdraw(harrys_account, 1000);
     // Insufficient funds.
     // Now harrys_account is 890
```

Parameter Modifications are Local

Here is a first attempt:

```
void withdraw(double balance, double amount)
{
   const double PENALTY = 10;
   if (balance >= amount)
   {
      balance = balance - amount;
   }
   else
   {
      balance = balance - PENALTY;
   }
}
```

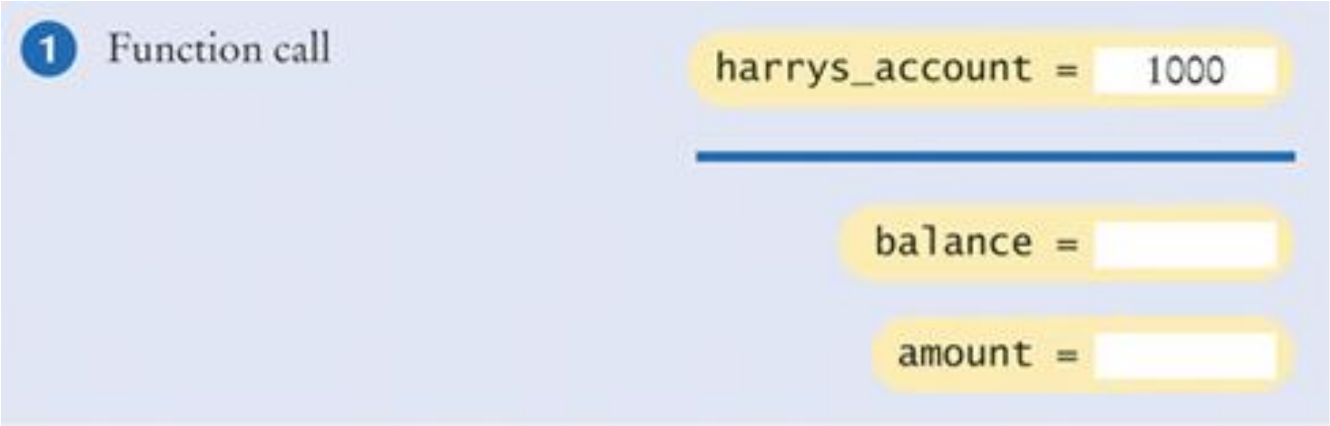
But this doesn't work.

Parameter Modifications are Local

What is actually happening?

Let's call the function passing in 100 to be taken from **`harrys_account`**.

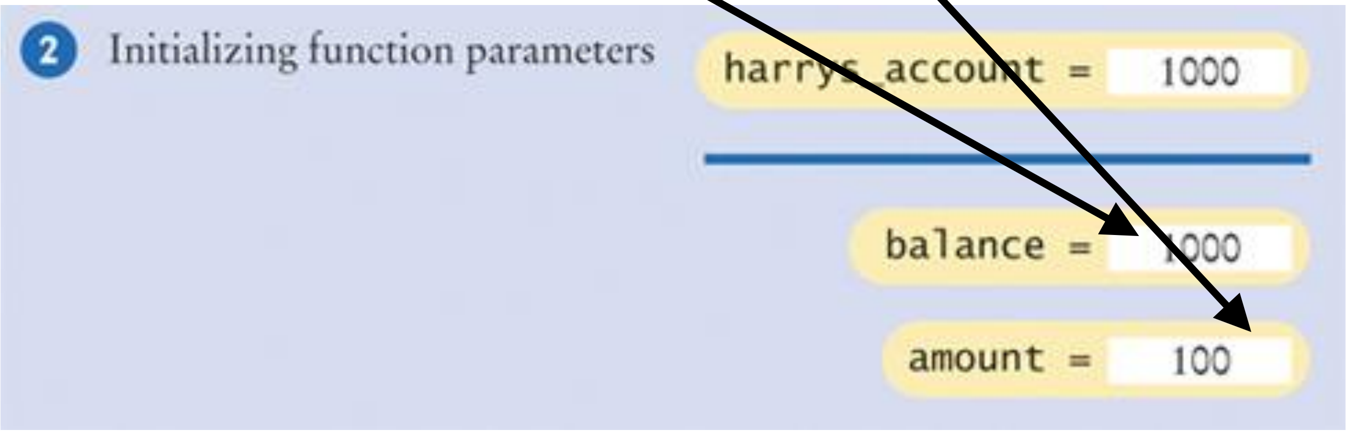
```
double harrys_account = 1000;

withdraw(harrys_account, 100);
```



Parameter Modifications are Local

The local variables, consts, and value parameters are initialized.

```
double harrys_account = 1000;
...
withdraw(harrys_account, 100);
...
void withdraw(double balance, double amount)
{
   const int PENALTY = 10;
```



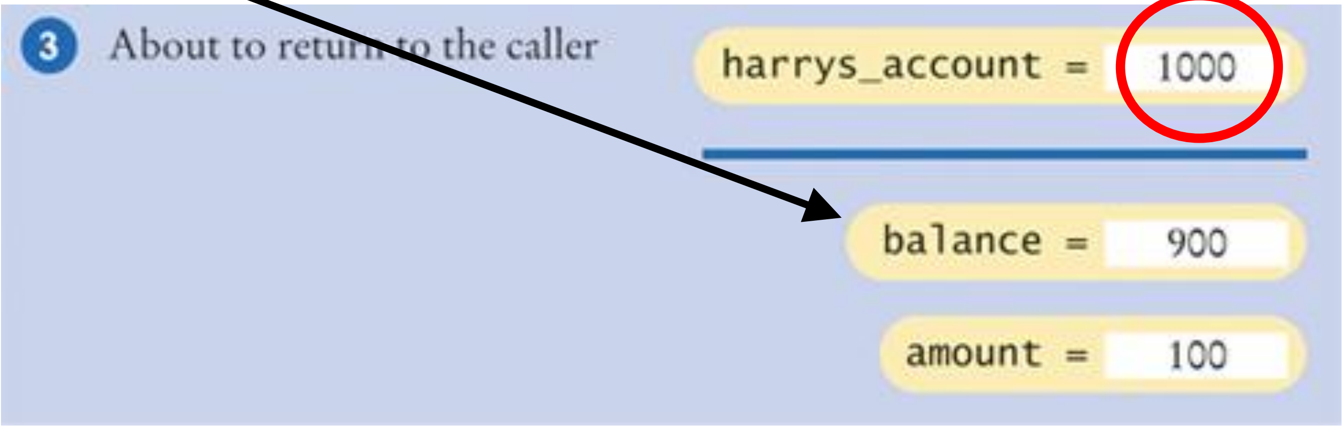
Parameter Modifications are Local

The test is **`true`**, the *LOCAL* variable **`balance`** is updated

```
double harrys_account = 1000;
...
    if (balance >= amount)
    {
        balance = balance - amount;
    }
```

NOTHING happens to **`harrys_account`** because it is a separate variable (in a different scope)

!



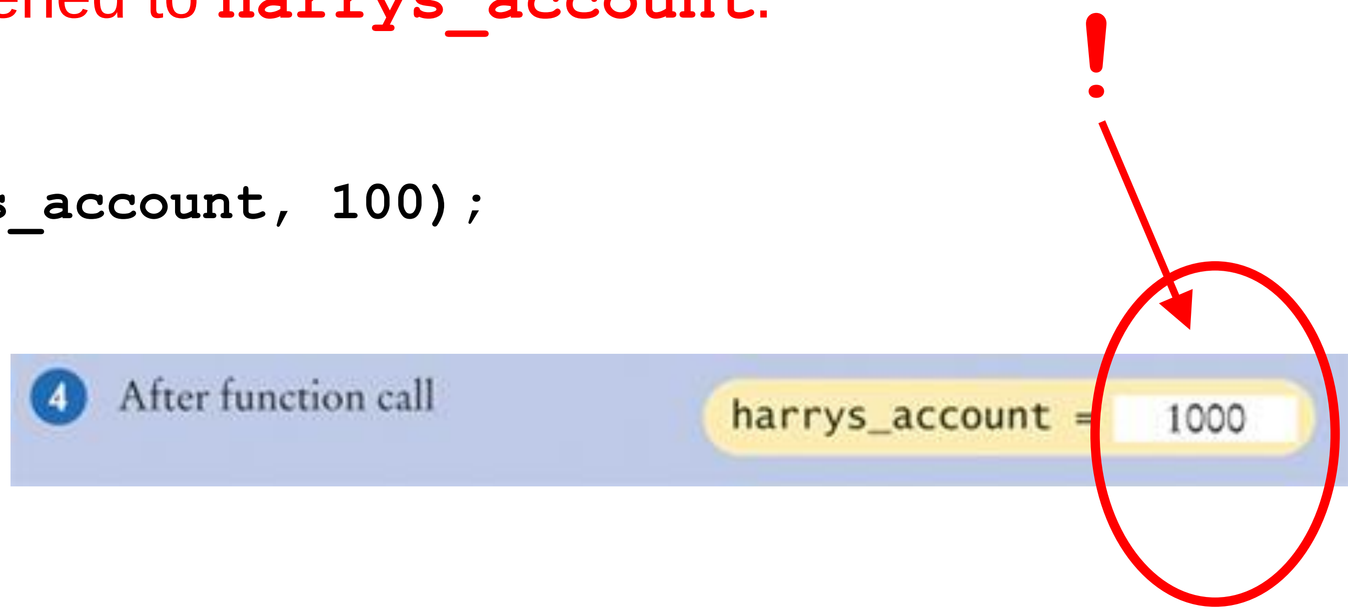
Parameter Modifications are Local

The function call has ended.

Local names in the function are gone and…

NOTHING happened to **`harrys_account`**.

```
withdraw(harrys_account, 100);
```



Parameter Pass by Value

```c
void withdraw(double balance, double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}
void deposit(double balance, double amount)
{
   balance = balance + amount;
}
int main()
{
   double balance = 10000;
   withdraw(balance, 1000);
   deposit(balance, 200);
   printf("%d", balance);
   return 0;
}
```

Parameter Pass by Value

Once we cover pointers, we will see that we can actually modify the value of parameters, and the effect is not local! Pass by reference!

Stay tuned!

Example: Game of Chance

- A dice game known as “craps”. The rules:
  - A player rolls two dice.
  - Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots.
  - After the dice have come to rest, the sum of the spots on the two upward faces is calculated.
    - If the sum is 7 or 11 on the first throw, the player wins.
    - If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins).
    - If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.”
  - To win, you must continue rolling the dice until you “make your point.” → sum of the dice = your point
  - The player loses by rolling a 7 before making the point.

Example

- We will write a function *rollDice* to throw the two dice and take their sum.

  (we did this for one die in the previous lecture)

```
int rollDice()
{
      int die1, die2, workSum;

      die1 = 1 + (rand()%6);
      die2 = 1 + (rand()%6);
      diceSum = die1 + die2;

      printf("Player rolled %d + %d = %d\n",
             die1, die2, diceSum);
      return diceSum;
}
```

Random Number Generation

- **rand ( ) function**
  - Included in <stdlib.h>
  - Returns a random integer number between 0 and RAND_MAX (32767)

    i = rand();

- Scaling
  - To get a random number between 1 and n

    1 + (rand() % n)

    - rand() % n   returns a number between 0 and n - 1
    - Add 1 to make the random number between 1 and n

    Example:

    1 + (rand() % 6)

    Number between 1 and 6

Random Number Generation

- **srand() function**
  - Included in <stdlib.h>
  - Takes an integer **seed** and initializes the random generator

    srand(*seed* **);**

  - Used to generate different random sequence for every rand() function call.
  - If seed is the same, it generates the same random sequence in every run of the program.

  - **srand(time(NULL))**; //Included in <time.h>
    - time(NULL)
      - Returns the time (in seconds) at which the program was executed
      - “Randomizes" the seed which guarantees complete randomness

Complete Program

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

enum Status {CONTINUE, WON, LOST};

int rollDice(); //function prototype

int main()
{
        int sum;
        int myPoint;
        enum Status gameStatus;

        srand(time(NULL));
        sum = rollDice();
```

Complete Program

- If the sum is 7 or 11 on the first throw, the player wins.
- If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins).
- If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.”

```
switch(sum){
        case 7:
        case 11:
                gameStatus = WON;
                break;
        case 2:
        case 3:
        case 12:
                gameStatus = LOST;
                break;
        default:
                gameStatus = CONTINUE;
                myPoint = sum;
                printf("Point is %d\n", myPoint);
                break;
}
```

Complete Program

To win, you must continue rolling the dice until you "make your point."
The player loses by rolling a 7 before making the point.

```
        while(gameStatus == CONTINUE)
        {
                sum = rollDice();
                if (sum == myPoint)
                        gameStatus = WON;
                else{
                        if (sum == 7){
                                gameStatus = LOST;
                        }
                }
        }
        if (gameStatus == WON){
                printf("Player wins\n");
        }
        else{
                printf("Player loses\n");
        }
    } //end of main
```

Enumeration constants

- The player may win or lose on the first roll, or may win or lose on any subsequent roll.
- Variable gameStatus, defined to be of a new type—enum Status—stores the current status.
- Line 8 creates a programmer-defined type called an enumeration.

```
enum Status gameStatus;
```

- An enumeration, introduced by the keyword enum, is a set of integer constants represented by identifiers.
- Enumeration constants are sometimes called symbolic constants.
- Values in an enum start with 0 and are incremented by 1.

Recursion

- Recursive functions
  - **Functions that call themselves are recursive.**
  - Can only solve a base case.
  - Divide a problem up into
    - What it can do **(base case) (a.k.a. termination part)**
    - What it cannot do **(recursive case)**
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved.
    - Gets plugged in, works its way up and solves whole problem.
    - Make sure that base case is reachable at some point!

Example: Recursive Factorial calculation

- `5! = 5 * 4 * 3 * 2 * 1`

- Notice that

  `5! = 5 * 4!`

  `4! = 4 * 3!`

  `...`

- Can compute factorials recursively
- Solve base case (`1! = 1 and 0! = 1`) then plug in

  `2! = 2 * 1! = 2 * 1 = 2`

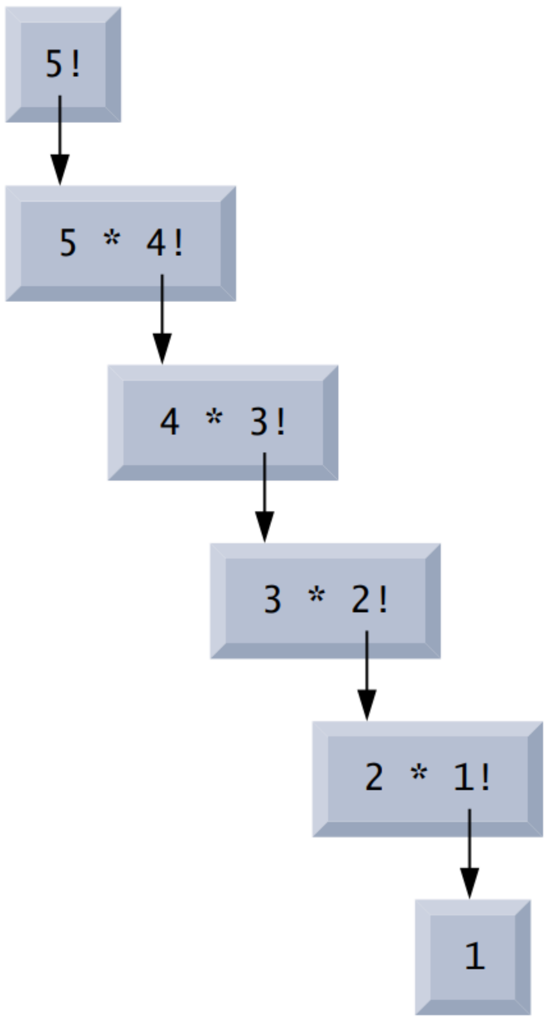
  `3! = 3 * 2! = 3 * 2 = 6`

Example: Recursive Factorial calculation

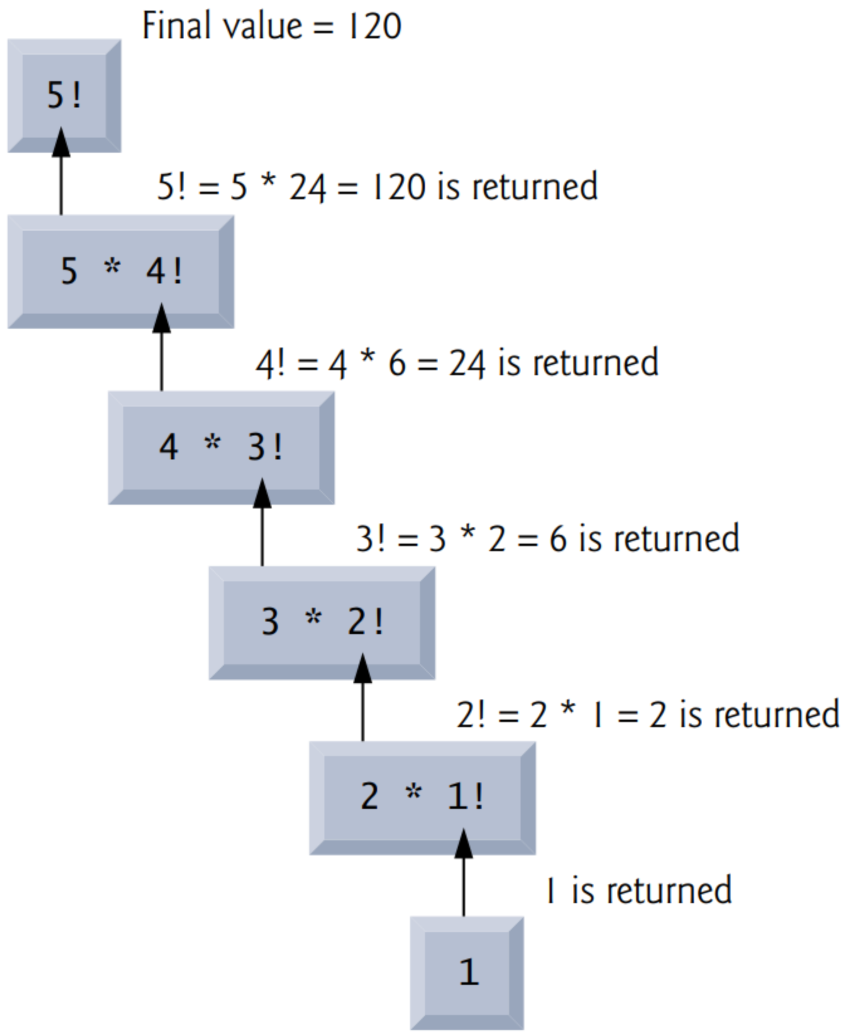
- The factorial function *f (n) = n! can be defined recursively as follows:*

$$
f(n) = \begin{cases} 1 & \text{if } n <= 1 \quad \textbf{(base case)} \\ n * f(n-1) & \text{otherwise} \quad \textbf{(recursive case)} \end{cases}
$$

Example: Recursive Factorial calculation



(a) Sequence of recursive calls



(b) Values returned from each recursive call

Example: Recursive factorial function

Part 1 of 2

```
/* Recursive factorial function */
#include <stdio.h>

int factorial( int number ); // function prototype

int main()
{
   int i; // counter

   /* loop 10 times; during each iteration, calculate
      factorial( i ) and display result */
   for ( i = 1; i <= 10; i++ ) {
      printf( "%2d! = %d\n", i, factorial( i ) );
   }

} // end main
```

%2d
Integer number is displayed in a field of 2 width.

Example: Recursive factorial function

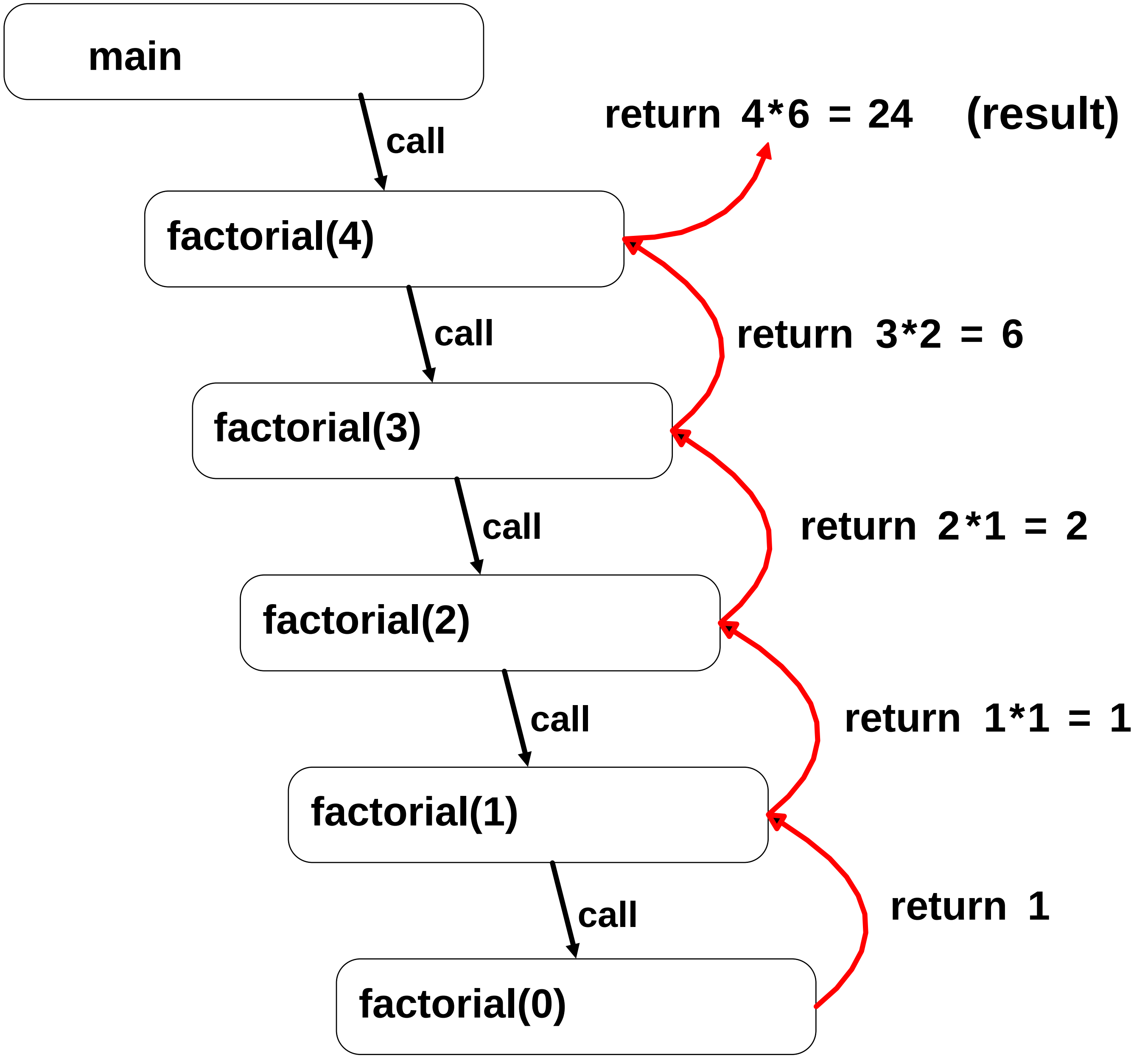
Part 2 of 2

```
/* recursive definition of function factorial */
int factorial( int number )
{
   // base case
   if ( number <= 1 ) {
      return 1;
   } // end if
   else { // recursive step
      return ( number * factorial(number - 1));
   } // end else

} // end function factorial
```

Program Output

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Example: Call sequence of recursive factorial



Non-recursive (Iterative) Factorial function

```
int factorial( int number )
{
   int i; // loop counter
   int result = 1;

   for (i=1 ; i <= number ; i++)
   {
      result = result * i ;
   }

   return result;
} // end function factorial
```

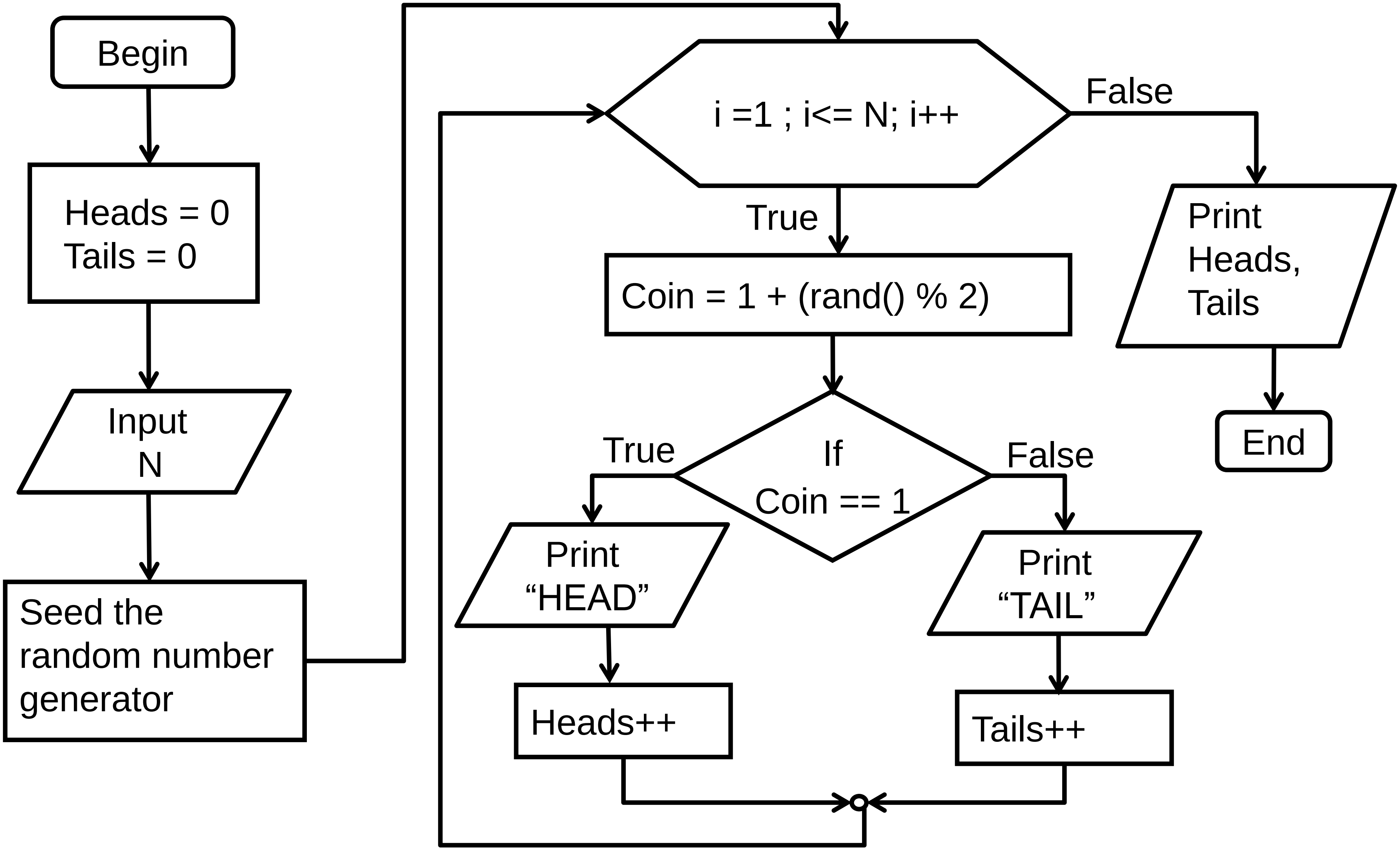
Avoid Infinite Recursion

- If the termination condition of a recursive function call is not designed properly, then the program may run inifinitely.
- The operating system can detect and stop an infinitely recursive program by displaying a run-time error message such as **"Stack Overflow".**

```
#include <stdio.h>

void f();   // Function prototype

int main()
{
   printf("Program started\n");
   f();  // First call of function f
   printf("Program finished\n");
}

void f()
{
  printf("Hello\n");
  f(); // Recursive call of function f (infinite)
}
```

Extra Self Study Examples

Example: Coin Toss Simulation



Example: Coin Toss Simulation

```c
#include <stdio.h>
#include <stdlib.h> // srand(),rand()
#include <time.h>   // time()

int main()
{
    int n;                  // Number of simulations
    int i;                  // Loop counter
    int coin;               // Random number (1 or 2)
    int heads = 0, tails = 0;   // Counters for heads and tails

    printf("Enter how many times the simulation will be done : ");
    scanf("%d", &n);

    srand(time(NULL));  // Seed the random number generator

```

Part 1 of 2

Example: Coin Toss Simulation

Part 2 of 2

```
for (i = 1;  i <= n;  i++)
{
        // Generate a random number (1 or 2)
        coin = 1 + (rand() % 2);

        if (coin == 1) {
            printf(" HEAD \n");
            heads++;
        } else {
            printf(" TAIL \n");
            tails++;
        }
}

   printf("Count of heads: %d   Percent: %.f \n",
           heads, (100.0 * heads) / n);

   printf("Count of tails: %d   Percent: %.f \n",
          tails, (100.0 * tails) / n);

} // end main
```

Program Output

```
Enter how many times the simulation will be done : 5

 TAIL

 TAIL

 HEAD

 TAIL

 HEAD

Count of heads: 2   Percent: 40
Count of tails: 3   Percent: 60
```

Example: Finding maximum of three numbers

- Program calls a function to find maximum of 3 numbers.

```
/* Finding the maximum of three integers */
#include <stdio.h>

int maximum( int x, int y, int z ); // function prototype

int main()
{
   int number1; // first integer
   int number2; // second integer
   int number3; // third integer

   printf( "Enter three integers: " );
   scanf( "%d %d %d", &number1, &number2, &number3 );

   /* number1, number2 and number3 are arguments
      to the maximum function call */
   printf( "Maximum is: %d\n", maximum(number1, number2, number3) );
} // end main
```

Part 1 of 2

```c
/* Function maximum definition */
// x, y and z are parameters
int maximum( int x, int y, int z )
{
   int max = x;     // initially assume x is largest

   if ( y > max ) { // if y is larger than max, assign y to max
      max = y;
   }

   if ( z > max ) { // if z is larger than max, assign z to max
      max = z;
   }

   return max;      // max is largest value

} // end function maximum
```

Program Outputs

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 85 22 17
Maximum is: 85
```

```
Enter three integers: 22 17 85
Maximum is: 85
```

Example: Variable scopes (Global and Local)

```c
#include <stdio.h>

int x = 2; // Global variable

/* function useGlobal modifies global variable x during each call */
void useGlobal()
{
   x = x * 10; //Global x
   printf("Global x is %d \n", x);
}

//------------------------------------------
int main()
{
   int x = 5; // local variable to main
   printf("Local x is %d \n\n", x );

   useGlobal();      // global x
} // end main
```

Program Output

```
Local x is 5

Global x is 20
```

Example: Displaying system date and time

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

int main()
{
  time_t  date_time;
  // Declare variable as time_t data type (long number)

  // Capture the current date and time as a long number
  time(&date_time);

  printf("%d \n\n", date_time); // Display as a long number

  // Convert the long number to a string definition and display it
  printf("%s \n", ctime(&date_time) );

}
```

Program Output

```
1445166873

Fri Jan 07 09:35:00 2024
```