

IS1201: Programming & Problem Solving

7. Functions & Recursion



Viraj Welgama

Functions

- A **function** is a block of code that performs a specific task.
- Dividing complex problem into small components makes program easy to understand and use.
- There are two types of functions in C programming:
 1. Standard library functions
 2. User defined functions

Standard Library Functions

- The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.
- These functions are defined in the header file. When you include the header file, these functions are available for use.
- For example:
 - The `printf()` is a standard library function to send formatted output to the screen (display output on the screen).
 - This function is defined in "`stdio.h`" header file.
 - There are other numerous library functions defined under "`stdio.h`", such as `scanf()`, `fprintf()`, `getchar()` etc.
 - Once you include "`stdio.h`" in your program, all these functions are available for use.

User-defined Functions

- C allows you to define functions according to your need. These functions are known as user-defined functions.
 - For example:
 - Suppose, you need to define a function to find the maximum number between 2 numbers or draw a circle when the radius is given.
 - Then you can create the following functions on your own.
1. `findMax(int x, int y)`
 2. `drawCircle(int r)`

Signature of a Function

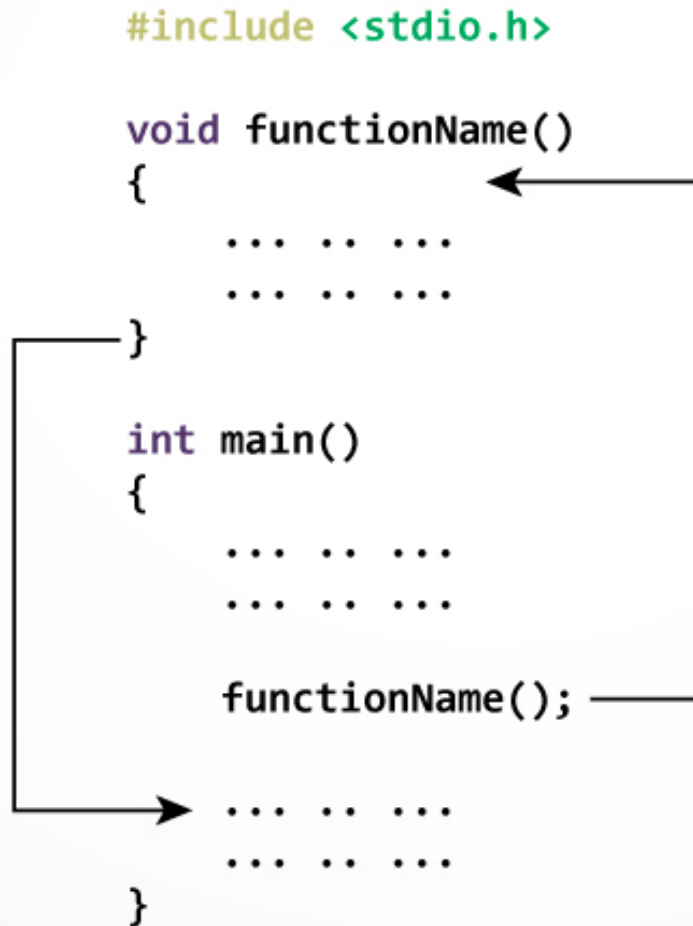
(return type) functionName (<arg1Type> <arg1Name>, <arg2Type> <arg2Name>, ...)



Types of User-defined Functions

1. Functions with **no arguments** and **no return value**
2. Functions with **no arguments** and a **return value**
3. Functions with **arguments** and **no return value**
4. Functions with **arguments** and a **return value**

How Function Works?



How Function Works?

```
# include <stdio.h>

void printMax(int x, int y) {
    printf("the large number is ");
    if(x>y)
        printf("%d\n", x);
    else
        printf("%d\n", y);
}

int main() {

    int num1, num2;

    printf("Enter your 1st number: ");
    scanf("%d", &num1);

    printf("Enter your 2nd number: ");
    scanf("%d", &num2);

    printMax(num1, num2);

    printf("-done!");
}
```


Advantages of Functions

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules.
4. Hence, a large project can be divided among many programmers.

A Function Prototype

- A function prototype is simply the declaration of a function that specifies function's name, parameters and return type.
 - It doesn't contain function body.
- A function prototype gives information to the compiler that the function may later be used in the program.

```
void printMax(int, int);
```

- is the function prototype which provides following information to the compiler:
 1. name of the function is printMax()
 2. return type of the function is void
 3. two arguments of type int are passed to the function

Passing Arguments

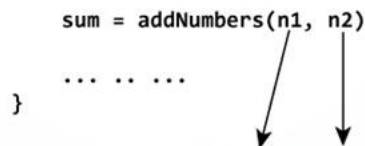
- In programming, argument refers to the variable passed to the function.
 - In the above example, two variables `num1` and `num2` are passed during function call.
- The parameters `x` and `y` accepts the passed arguments in the function definition.
- These arguments are called formal parameters of the function.

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```



The diagram illustrates the passing of arguments from the `main` function to the `addNumbers` function. Two arrows originate from the parameters `n1` and `n2` in the function call `sum = addNumbers(n1, n2);` within the `main` function. These arrows point to the parameters `a` and `b` in the function definition `int addNumbers(int a, int b)`, demonstrating how the arguments are passed to the function's formal parameters.

Return statement

- The **return** statement terminates the execution of a function and returns a value to the calling function.
- The program control is transferred to the calling function after the **return** statement.
- Syntax:
return (expression);

```
return value;  
return x+y;
```

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```

sum = result

Example: return

```
int findMax(int, int);

int main() {

    int num1, num2, max;
    printf("Enter your 1st number: ");
    scanf("%d", &num1);

    printf("Enter your 2nd number: ");
    scanf("%d", &num2);

    max = findMax(num1, num2);

    printf("the maximum number is %d\n", max);
    printf("the power of the maximum number is %d\n", findMax(num1, num2) * findMax(num1, num2));

    printf("-done!");

}

int findMax(int x, int y) {
    if(x>y)
        return x;
    else
        return y;
}
```

Problems

1. Write a program to calculate area of a cylinder by using macros.
2. Company XYZ & Co. pays all its employees Rs.150 per normal working hour and Rs. 75 per OT hour. A typical employee works 40 (normal) and 20 (OT) hours per week has to pay 10% tax. Develop a program that determines the take home salary of an employee from the number of working hours and OT hours given.

Problems

3. Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer the people who can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of Rs 15.00 per ticket, 120 people attend a performance. Decreasing the price by 5 Rupees increases attendance by 20 and increasing the price by 5 Rupees decreases attendance by 20. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner Rs.500. Each attendee costs another 3 Rupees. The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

when we are confronted with such a situation, it is best to tease out the various dependencies one at a time:

Profit is the difference between revenue and costs.

The revenue is exclusively generated by the sale of tickets. It is the product of ticket price and number of attendees.

The costs consist of two parts: a fixed part (Rs.500) and a variable part (Rs. 3 per attendee) that depends on the number of attendees.

Write a program to solve this problem.

Recursions

- It is legal for one function to call another.
- It is also legal for a function **to call itself**.
 - like a snake that swallows its own tail, a function can call itself.
- It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.
- A function that calls itself is recursive; the process is called recursion.



Recursions: Example

- look at the following C function:

```
void countdown(int n){  
    printf("%d \n",n);  
    if(n>1)  
        countdown(n-1);  
    else  
        printf("Blast!!\n");  
}
```

- If **n** is less than or equal to one, it outputs the word, “Blast!!”
- Otherwise, it calls a function named **countdown()** itself by passing **n-1** as an argument.
- What if we call this function as **countdown(3)** ?

Recursions: Example

- The execution of `countdown(3)` begins with `n=3`, and prints `3` on the terminal. Since `n` is greater than `1`, it calls itself with `n-1` as the argument (as `countdown(2)`)
- The execution of `countdown(2)` begins with `n=2`, and prints `2` on the terminal. Since `n` is still greater than `1`, it calls itself with `n-1` as the argument (as `countdown(1)`)
- The execution of `countdown(1)` begins with `n=1`, and prints `1` on the terminal. Since now `n` is NOT greater than `1`, it prints “Blast!!” on the terminal and stop.

```
void countdown(int n){  
    printf("%d \n",n);  
    if(n>1)  
        countdown(n-1);  
    else  
        printf("Blast!!\n");  
}  
//countDown()
```

Recursions: Example

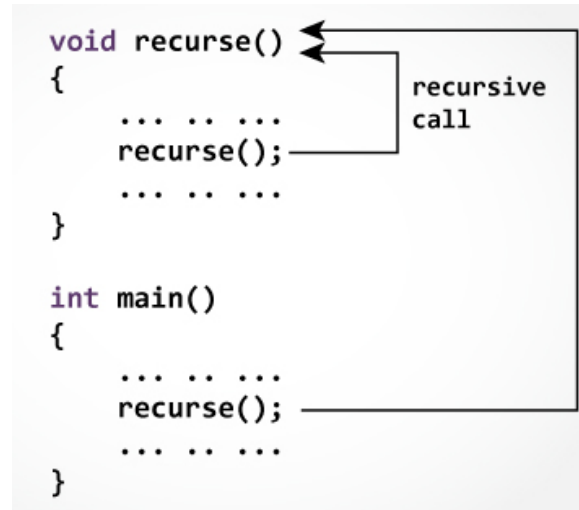
- The execution of `countdown(3)` begins with `n=3`, and prints `3` on the terminal. Since `n` is greater than `1`, it calls itself with `n-1` as the argument (as `countdown(2)`)
- The execution of `countdown(2)` begins with `n=2`, and prints `2` on the terminal. Since `n` is still greater than `1`, it calls itself with `n-1` as the argument (as `countdown(1)`)
- The execution of `countdown(1)` begins with `n=1`, and prints `1` on the terminal. Since now `n` is NOT greater than `1`, it prints “Blast!!” on the terminal and stop.

```
void countdown(int n){  
    printf("%d \n",n);  
    if(n>1)  
        countdown(n-1);  
    else  
        printf("Blast!!\n");  
} //countDown()
```

```
3  
2  
1  
Blast!!
```

Recursive Functions

- A function that calls itself is known as a **recursive function**. and, this technique is known as recursion.



- The recursion continues until some condition is met to prevent it.

Factorial Function

- Factorial $n = 1 \times 2 \times 3 \dots \times n$

Factorial Function

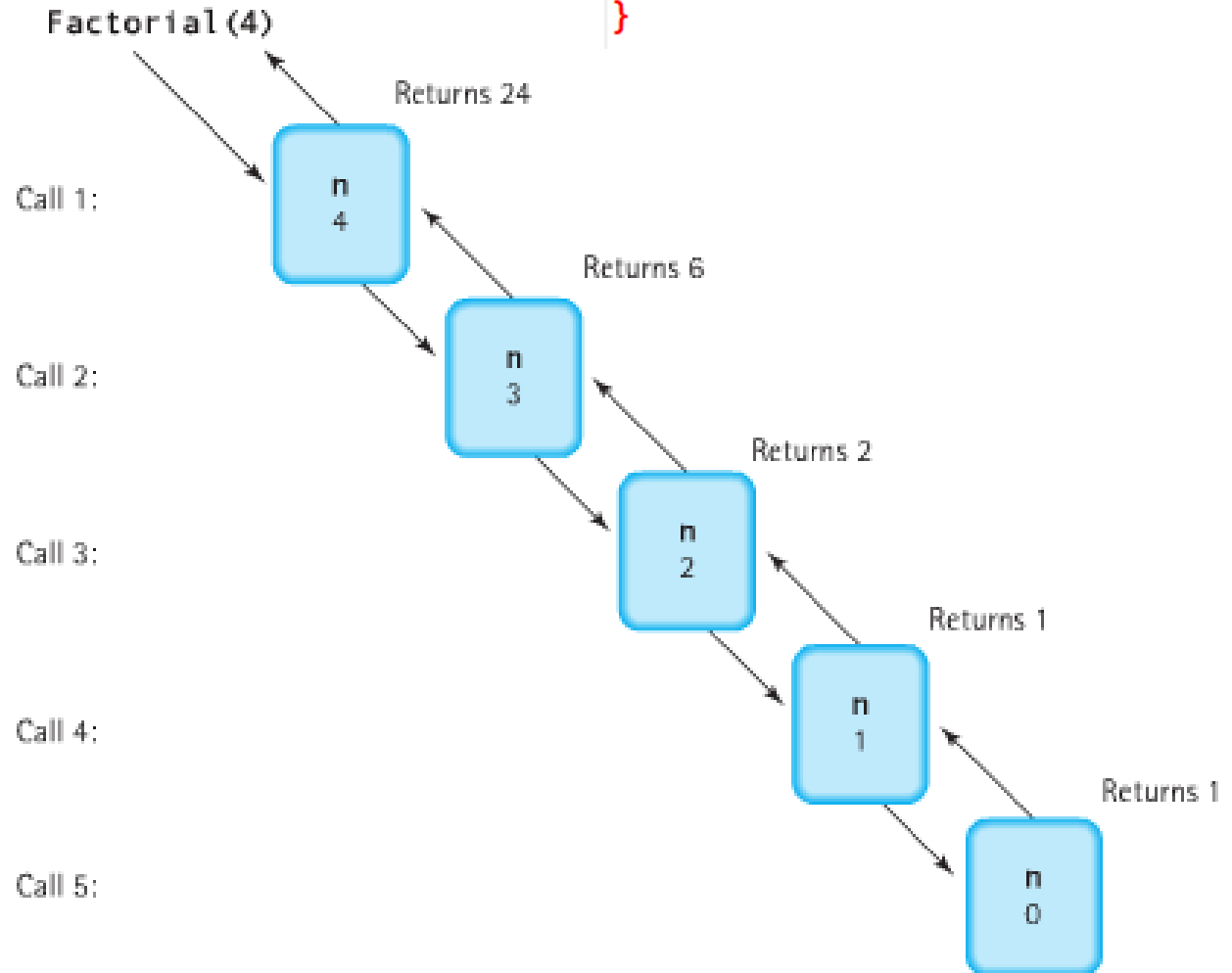
- Factorial $n = 1 \times 2 \times 3 \dots \times n$

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

Factorial Function

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

If we called it as
factorial(4)?



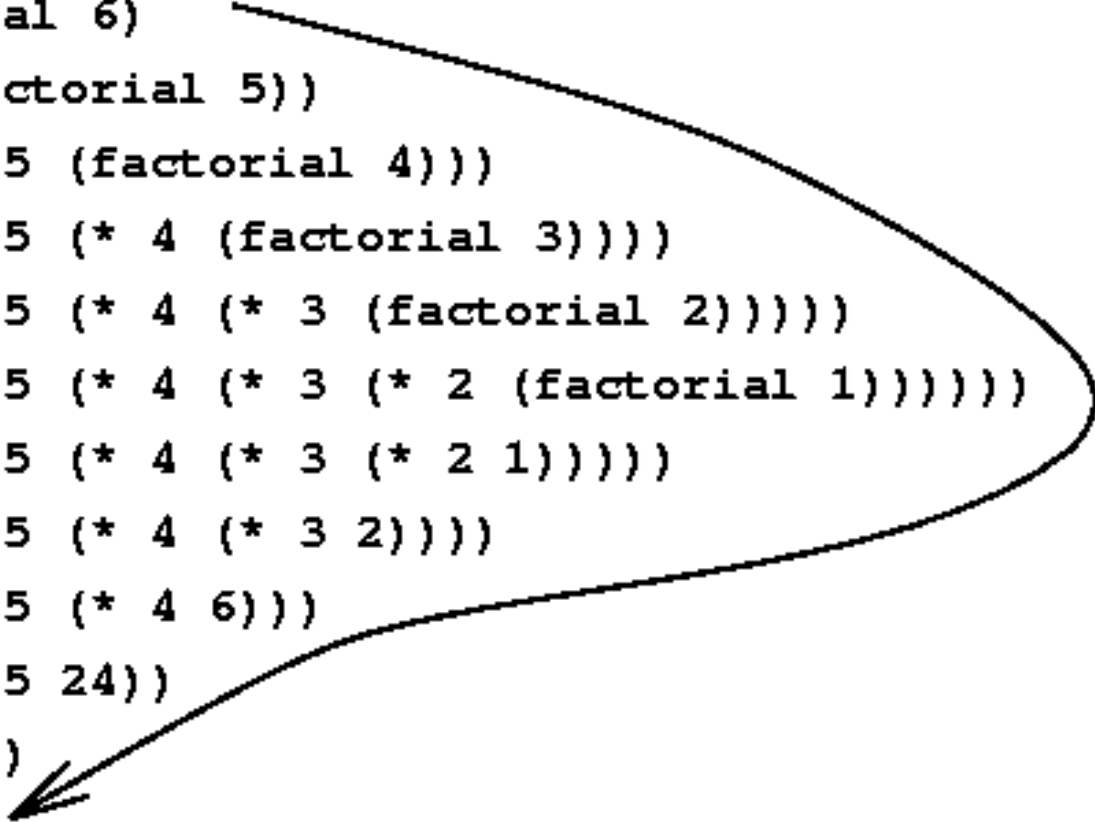
Factorial Function

If we called it as
`factorial(6)`?

Factorial Function

If we called it as
factorial(6)?

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

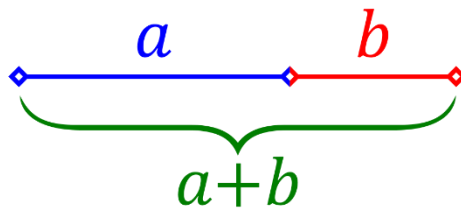


Infinite Recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates.
- This is known as infinite recursion, and it is generally not a good idea.

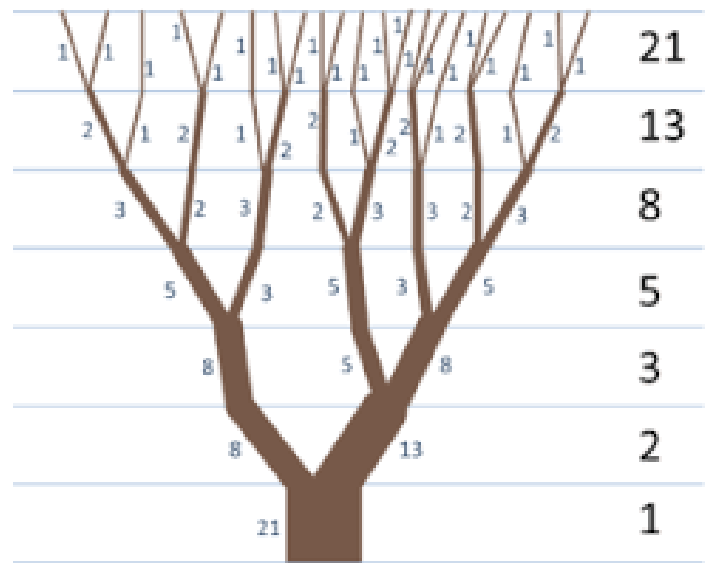
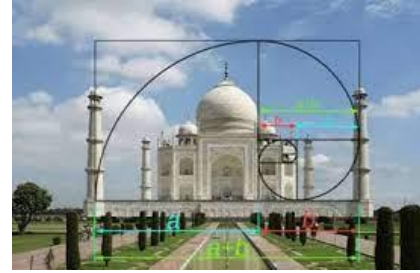
Fibonacci

- Fibonacci is a sequence, starting with 0 and 1 and then the **each number is the sum of the two preceding ones**.
 - i.e: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern, and the arrangement of a pine cone's bracts.
- Fibonacci numbers are strongly related to the golden ratio.



1 : 1.168

$a+b$ is to a as a is to b



Fibonacci

- After factorial, the most common example of a recursively defined mathematical function is Fibonacci
- i.e:
 - $\text{fibonacci}(0) = 0$
 - $\text{fibonacci}(1) = 1$
 - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$

Fibonacci

- After factorial, the most common example of a recursively defined mathematical function is Fibonacci
- i.e:
 - $\text{fibonacci}(0) = 0$
 - $\text{fibonacci}(1) = 1$
 - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$

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

Fibonacci

- After factorial, the most common example of a recursively defined mathematical function is Fibonacci
- i.e:
 - $\text{fibonacci}(0) = 0$
 - $\text{fibonacci}(1) = 1$
 - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$

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

```
int fibonacciPrint(int n) {
    for(int i=1; i<=n; i++)
        printf("%d, ", fibonacci(i));
} //fibonacciPrint
```

Pros & Cons of Recursion

- Recursion makes program elegant and cleaner.
- All algorithms can be defined recursively which makes it easier to visualize and prove.
- If the speed of the program is vital then, you should avoid using recursion.
- Recursions use more memory and are generally slow.
- Instead, you can use loop.

Problem

The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b ,

$$\text{then } \gcd(a, b) = \gcd(b, r).$$

As a base case, we can use $\gcd(a, 0) = a$.

Write a recursive functions to calculate GCD of any given two integers.