



University  
of Glasgow

# UESTC 1005 – Introductory Programming

## Lecture 4 – Functions

Dr Hasan T Abbas

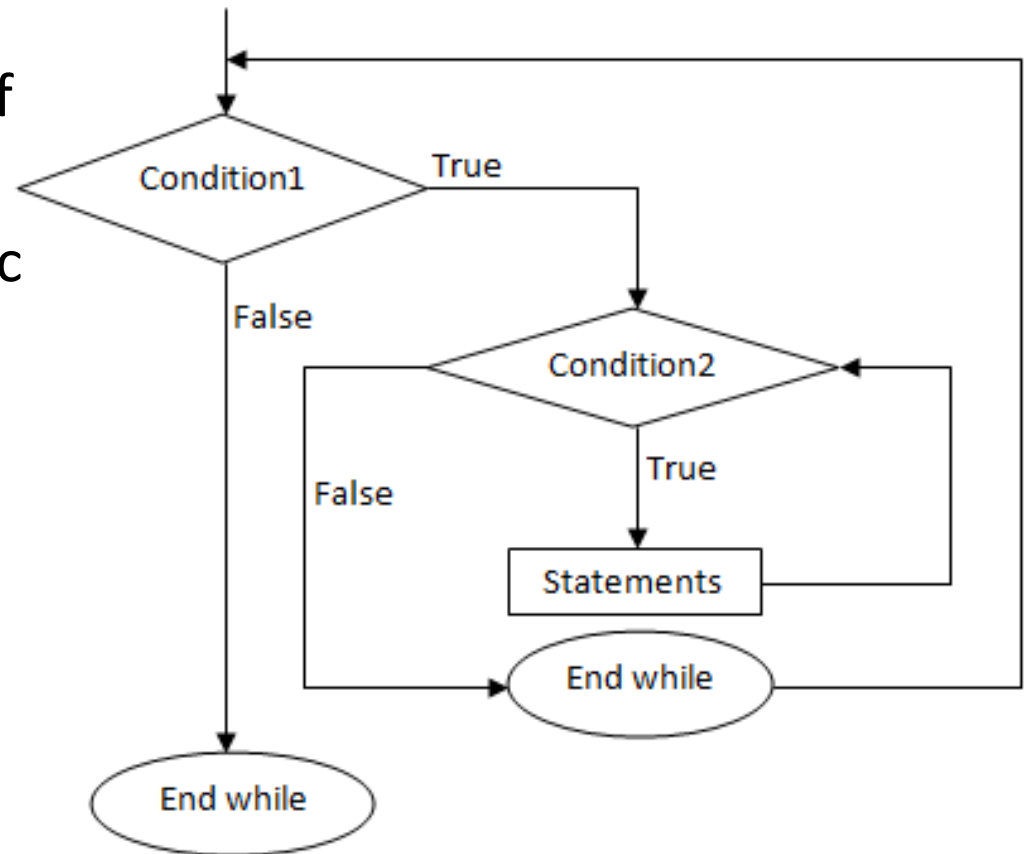
[Hasan.Abbas@glasgow.ac.uk](mailto:Hasan.Abbas@glasgow.ac.uk)

Fall 2019

Glasgow College – UESTC

# Nested Loops

- A loop inside another loop is called a nested loop.
- We can have any number of nested loops
- Extensively used in scientific computation
- We can program matrices (linear algebra), process images and so on...



# Nested Loops - Syntax

- Two level nested loop contains OUTER and INNER loops
- If outer loop has N iterations
  - Inner loop has M iterations
- In total we will have  $N * M$  iterations

```
for (expr1 ; expr2 ; expr3){  
    for (expr4 ; expr5 ; expr6){  
        expression statements;  
    }  
}
```

# Nested Loops - Example

- Print Numbers in a shape

```
#include <stdio.h>
int main()
{
    int i=1,j;
    for (i = 1; i <= 5; i++)
    {
        for (j=1; j <= I; j++ )
        {
            printf("%d ",j);
            j++;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

# Nested Loops - Question

- Print Numbers .....

```
#include <stdio.h>
int main () {
    int i, j;
    for(i = 2; i<15; i++) {
        for(j = 2; j <= i/j; j++){
            if( !(i%j) ){
                break;
            }
        }
        if(j > (i/j)){
            printf("%d\n", i);
        }
    }
    return 0;
}
```

# Functions

# Functions

- A group of statements that perform a specific task together.
- In mathematics, functions generally have inputs (arguments) and outputs
  - For example,  $y = \sin\left(\frac{\pi}{4}\right) = 0.707$
- However, in C programming language, we have functions that:
  - May or may not have inputs (arguments)
  - May or may not have an output (return)
- We give a name to the function
  - For example `main()`
- Functions encourage modular programming
  - Building blocks of C program
  - Easier to understand and modify programs

# Function Structure in C

- A function is composed of 4 elements (+ *the parentheses*)

```
return_type function_name (arguments/parameter list)
{
    function body;
}
```

```
double average (double a, double b)
{
    return (a + b)/2;
}
```



# Function Structure in C

- **return\_type**: A function may return a value. The **return\_type** determines the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return\_type** is the **void**.
  - e.g., `void powr (int a)`
- **function\_name**: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
  - Follow same rules as for variable names.
- **Argument/Parameter list**: When a function is invoked/called, we pass a value as an argument. This value is referred to as the actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
  - e.g., `int powr (int a) , void hello_word(void)`
- **Function Body**: The function body contains a collection of statements that define what the function does. Normally delimited by braces but can be omitted if it contains only one line (not recommended).

# Defining a C Function

- Choose a function name

Bingo, printer, powr

Not Allowed: **2pac**, **#swag**

- Function arguments (if any)

a. Data type of arguments: func( int a, char b,  
double c)

b. Number of arguments: func( void)

c. Order of arguments

- Function return type

- May be any variable data type
- Can even be void

# Function Prototype

- In a C program, when there are functions other than `main()`, we need to tell the compiler that **there are** other functions.
- A function prototype provides a *complete description* how that function needs to be called/invoked in a program
  - For example, `double average (double, double);`
- Notice there are no variable names in the arguments list; just the type.
- Also a semicolon at the end.
- Function prototype must come **BEFORE** the function is called.

```
double average (double , double ); // prototype
int main(){
    double a = 2.5, b = 3.5;
    // function call below
    printf("The average of %lf and %lf is %lf",a,b, average(a,b));
    return 0;
}
double average (double x, double y){ // function definition
    return (x + y)/2;
}
```

# Passing by Value - Functions

Slight Difference between Arguments and Parameters

**Arguments** expressions that appear in **function calls**

**Parameters** are dummy names that appear in the **function definition**

**Pass by value:** when we call a function, each argument is evaluated, and its value is assigned to the corresponding parameter. In other words, we only supply a copy of the variable to a function.

The execution of the function **doesn't affect the argument**

# *Passing by Value* - Example

A function that calculates the power

```
int powr(int, int);

int main()
{
    int a = 4;
    int b = 2;

    printf("%d\n%d\n", powr(3,b), powr(a,b));
}

int powr(int base, int n)
{ //nth power of base
    int i, p=1;
    for (i=1; i<=n; i++) p=p*base;
    return p;
}
```

- two approaches to structure :

```
printf("%d\n%d\n", powr(3,b),powr(a,b));
```

copies

3

2

4

2

```
int powr(int base, int n) { //nth power of base
    int i, p=1;
    for (i=1; i<=n, i++) p=p*base;
    return p;
}
```

```
graph TD
    subgraph printf_stmt [printf("%d\n%d\n", ...)]
        direction LR
        P1[3]
        P2[2]
        P3[4]
        P4[2]
    end

    subgraph powr_func [int powr(int base, int n)]
        direction TB
        P1 --> B1[base]
        P2 --> N1[n]
        P3 --> B2[base]
        P4 --> N2[n]
        R1[return p]
    end

    R1 --> P1
    R2[return p] --> P3
```

# Recursion - Functions



A function is recursive when it calls itself

example: factorial function ( $N!$ )

```
int factorial (int); // prototype
int main(){
    int a = 6;
    // function call below
    printf("The factorial of %d is %d",a, factorial(a));
    return 0;
}
int factorial (int x){ // function definition
    if(n <= 1)
        return 1;
    else
        return x * factorial( x - 1); // recursive call
}
```

Recursion operates in **Last-in First-out (LIFO)** format

# System Functions

Besides the functions defined by the user, C has many built-in functions

Function	Description	Example
<code>sqrt(x)</code>	square root of $x$	<code>sqrt(900.0)</code> is 30.0 <code>sqrt(9.0)</code> is 3.0
<code>cbrt(x)</code>	cube root of $x$ (C99 and C11 only)	<code>cbrt(27.0)</code> is 3.0 <code>cbrt(-8.0)</code> is -2.0
<code>exp(x)</code>	exponential function $e^x$	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>log(x)</code>	natural logarithm of $x$ (base $e$ )	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of $x$ (base 10)	<code>log10(1.0)</code> is 0.0 <code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>fabs(x)</code>	absolute value of $x$ as a floating-point number	<code>fabs(13.5)</code> is 13.5 <code>fabs(0.0)</code> is 0.0 <code>fabs(-13.5)</code> is 13.5
<code>ceil(x)</code>	rounds $x$ to the smallest integer not less than $x$	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0

List of some functions in `<math.h>`



# System Functions

To include system functions, we include the **header files** using the preprocessor directive `#include`

Header	Explanation
<code>&lt;assert.h&gt;</code>	Contains information for adding diagnostics that aid program debugging.
<code>&lt;ctype.h&gt;</code>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<code>&lt;errno.h&gt;</code>	Defines macros that are useful for reporting error conditions.
<code>&lt;float.h&gt;</code>	Contains the floating-point size limits of the system.
<code>&lt;limits.h&gt;</code>	Contains the integral size limits of the system.
<code>&lt;locale.h&gt;</code>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
<code>&lt;math.h&gt;</code>	Contains function prototypes for math library functions.
<code>&lt;setjmp.h&gt;</code>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<code>&lt;signal.h&gt;</code>	Contains function prototypes and macros to handle various conditions that may arise during program execution.

List of some header files (Deitel Fig. 5.10)

# Random Number Generation

- The C library `<stdlib.h>` has built-in functions for random number generation.
- `rand( )` generates an integer between 0 and 32767
- `rand( )` generates **different numbers in multiple iterations** during one execution of a program.
- However, we get the **same numbers** if we execute the **program again!**
- To completely randomize the process, we use the `srand( . )` function from the `<stdlib.h>` library
- It prevents the same numbers to be generated every time we run the code.