

# EEE101: C Programming & Software Engineering I

## Lecture 5: Functions

Dr. Rui Lin/Dr. Mark Leach

Office: EE512/EE510

Email: [rui.lin/mark.leach@xjtlu.edu.cn](mailto:rui.lin/mark.leach@xjtlu.edu.cn)

Dept. of EEE XJTLU

# Outline of Today's Lecture (5)

- What are functions?
- Library functions
- User-defined functions
- Variables and variable scope
- Calling functions
- Recursion
- **#define** Macros
- Random number generation function

**but....first....**

# Assignment 2

- Will be distributed today...
- Worth 10%
- Assessing/developing your understanding and ability to use operations and flow control
- Due date is 22/10/2015 (Thursday week 6)
- Prospective assessment calendar:

Assignment 2	12/10/2015 (Mon. Wk5)	22/10/2015 (Thurs. Wk6)
Assignment 3	23/10/2015 (Fri. Wk6)	9/11/2015 (Mon. Wk9)
Assignment 4	9/11/2015 (Mon. Wk9)	23/11/2015 (Mon. Wk11)
Assignment 5	23/11/2015 (Mon Wk11)	7/12/2015 (Mon. Wk13)
Assignment 6	16/11/2015 (Wk10)	23/12/2015 (Wk15)

# Introduction

- Computer programs that solve real-world problems are usually made of smaller manageable pieces of code or **modules**.
- This is achieved by the so-called “**divide and conquer**” technique.
- In C, modules are called functions
- Modular programming favours **maintenance, code reuse and efficiency**, using library functions

# What is a function?

- A **self-contained** unit of computer code
- Designed to accomplish a particular task

A function can:

perform an action – **printf()** displays data

provide a value – **sqrt()** computes the square  
root of a positive real number

# Why use functions?

- The enable high-level **abstraction**
- Avoid repetition of code
- **Modular** programs are:
  - Easier to read
  - Easier to maintain
  - Easier to test and fix

# Math Library Functions

- Contains functions for performing some common mathematical calculations
- Accessed using the header file `#include<math.h>`

Function	Description
<code>sqrt(x)</code>	square root of x
<code>exp(x)</code>	exponential of x
<code>log(x)</code>	natural logarithm of x
<code>fabs(x)</code>	absolute value of x
<code>pow(x,y)</code>	x raised to the power y
<code>cos(x)</code>	cosine of x

# Character Library Functions

- Contains useful functions to perform tests and manipulations of character data.
- Accessed using the header file `#include <ctype.h>`

Function	Description
----------	-------------

<code>isdigit(x)</code>	returns 1 <b>if</b> x is a digit (0-9), <b>else</b> 0
-------------------------	-------------------------------------------------------

<code>isalpha(x)</code>	returns 1 <b>if</b> x is a letter (Aa-Zz) , <b>else</b> 0
-------------------------	-----------------------------------------------------------

<code>isalnum(x)</code>	returns 1 <b>if</b> x is a digit or letter, <b>else</b> 0
-------------------------	-----------------------------------------------------------

<code>islower(x)</code>	returns 1 <b>if</b> x is a lowercase letter (a-z), <b>else</b> 0
-------------------------	------------------------------------------------------------------

<code>toupper(x)</code>	converts x to a lowercase letter <b>if</b> it is uppercase
-------------------------	------------------------------------------------------------



# User-Defined Functions

- So far, you have written programs consisting of a function called **main()**
- In addition to the functions in the libraries, you can write **your own customised functions**.
- You should try to limit functions you write to performing a **single, well defined task**.
- You should choose a **name** for the function that **reflects** the **task**.

# A modular program

```
#include <stdio.h>
int square (int);      /*function prototype*/
int main (void){
    int x;
    for (x=1, x<=10; x++)
        {printf("%d ", square(x));
          printf ("\n");}
    return 0;
}                        /*End of main()*/

/* Function definition */
int square (int y) {
    return y*y;
}
```

# Function recipe

Step 1: Declare the function (like declaring a variable)

`int square(int);`

Determines the type of values that the function receives (can be many), in this case an integer value is sent to the function

Determines what kind of value (if any), that the function returns (only one)

# Function recipe

Step 1: Declare the function (like declaring a variable)

```
int square(int);
```

Step 2: Define the function

```
int square(int y){  
    return y*y;  
}
```

Declares variable y



/\*write some code\*/

Step 3: Use the function

```
square(x);
```

Perform operations  
and return value



# Function Declarations

A function without any arguments or return value

**void** function-name(**void**);

A function with arguments and no return value:

**void** function-name(**int**, **float**, **char**,...,**int**);

A function with arguments and a return value:

**float** function-name(**float**, **char**,...,);

# Function Arguments

```
int multiply (int x, int y)
{
    int z;
    z = x * y;
    return z;
}
```

x and y are formal **arguments** or **parameters** (inputs)

x and y are **private** to the function multiply

x and y are assigned values when the function is called

```
int z=5, x;
x = multiply(z,25);
```

**5** and **25** are the **actual arguments** sent to multiply

# Function Argument Order

As well as constant values and variables, functions (with **return** values) may be placed in the function call as an argument:

```
x=multiply(add(x));
```

You can also perform operations in the function call:

```
x=multiply(x+y);
```

Whatever operations you place in the function call, be careful the result must be **unaffected** by the **processing order**, as it is **not defined in C**

```
x=multiply(add(x)-add(x));
```

# Function **return** Value

Functions can **return** a value (output)

```
int multiply (int x, int y)
{
    int z;
    z = x*y;
    return z;
}
```

The **return** keyword indicates what is returned. The returned value should be the same type as the function type – otherwise automatic casting occurs

When **return** is reached, the function terminates.



# Declaring or Defining a Function

**Do not** confuse **declaring** or **defining** of functions

## **Function declaration**

Informs compiler of the function name, arguments and return values. Allows the compiler to check the variable types match. If there are no arguments or return values, use **void** otherwise the compiler does not check!

## **Function definition**

Provides the actual code – function body and uses computer memory

# Variables - Global or Local?

Variables can be declared **inside** or **outside** of a function.

```
int x;  
main()  
{  
  int y;  
  ...
```

x is declared **outside** main(), it is **global**.  
That means it is **visible everywhere**

y is declared **inside** main(), it is **local**.  
That means it is **only visible inside** main()

- Global variables exist for the entire program run time.
- The scope of a variable describes where it can be referred to by its name.

# Variables - Global or **Local**?

A variable declared **inside** a function (or block) is called a **local** (or automatic) variable.

Local variables are only known and can only be accessed by name inside the function where it is declared.

Local variables are **created** when the function is called and **destroyed** when the function ends. Note:

- They do not maintain values between function calls.
- Arguments received by a function are local to that function.

# Global vs. Local

- Initially it may seem easier to use only global variables since they are available everywhere **(do not need to be passed between functions)**
- Global variables can make programs difficult to read
- Global variables mean less efficient memory use
- Always try to use local variables and pass parameters.

# Local Variables and Scope

- The extent (or range) of the **visibility** of a variable within a program is often called its **scope**

```
void function1(int x0, int y0, int w, int h) {  
    int xr = x0 + w - 1;  
}  
void function2() {  
    printf ("Left pixel=%d Right pixel=%d\n", x0, xr);  
}    /* incorrect – x0 and xr are out of scope*/
```

- Variables x0 and xr are **local** to function1()
- They are **out of scope** from function2()

# Block Scope...{ }

- Variables may be declared following the beginning of any compound statement {...}
- Variables local to a block hide external variables with the same name:

```
main(){
```

```
int i=0, n=9;
```

```
if (n==9){
```

```
    int i=10;
```

```
    ...}
```

```
}
```

these are NOT the same variable

inside the if block this is used

Outside of the block only this i exists

# Scope Rules - Summary

**Global variables:** Variables declared outside of any function, they retain their values throughout program execution (i.e. use memory)

Variables declared inside a block are **local** and have **block scope**.

Variables declared **outside ALL** functions **have file scope**. Functions have file scope, i.e. can be called anywhere in a file.

# static Scope

The key word **static** applied to a variable

```
static int x=0;
```

- Limits the scope of a global variable or function to the current source code file.
- Once a **static** variable is declared it is alive till the file it is in ends.
- A **static** variable declared inside a block remains in existence and retains its value even after the block ends. i.e. if the block is re-entered, the previous value will remain.



# Calling Functions...**By Value**

In C, when functions are called with arguments, these are **call by value**: a copy of the data is passed to the function. The function **CANNOT** modify the original variable value.

## Calling function

```
int y=10,z;  
z=test(y);  
printf("y=%d",y);
```

## Function Code

```
int test (int x)  
{  
    return x*x;  
}
```


**Q. How can the calling function be changed to simulate call by reference (i.e. update the calling variable)?**

# Calling Functions...**By Value**

In C, when functions are called with arguments, these are **call by value**: a copy of the data is passed to the function. The function **CANNOT** modify the original variable value.

## Calling function

```
int y=10,z;  
y=test(y);  
printf("y=%d",y);
```



## Function Code

```
int test (int x)  
{  
    return x*x;  
}
```

**Reassign the return value to the original variable**

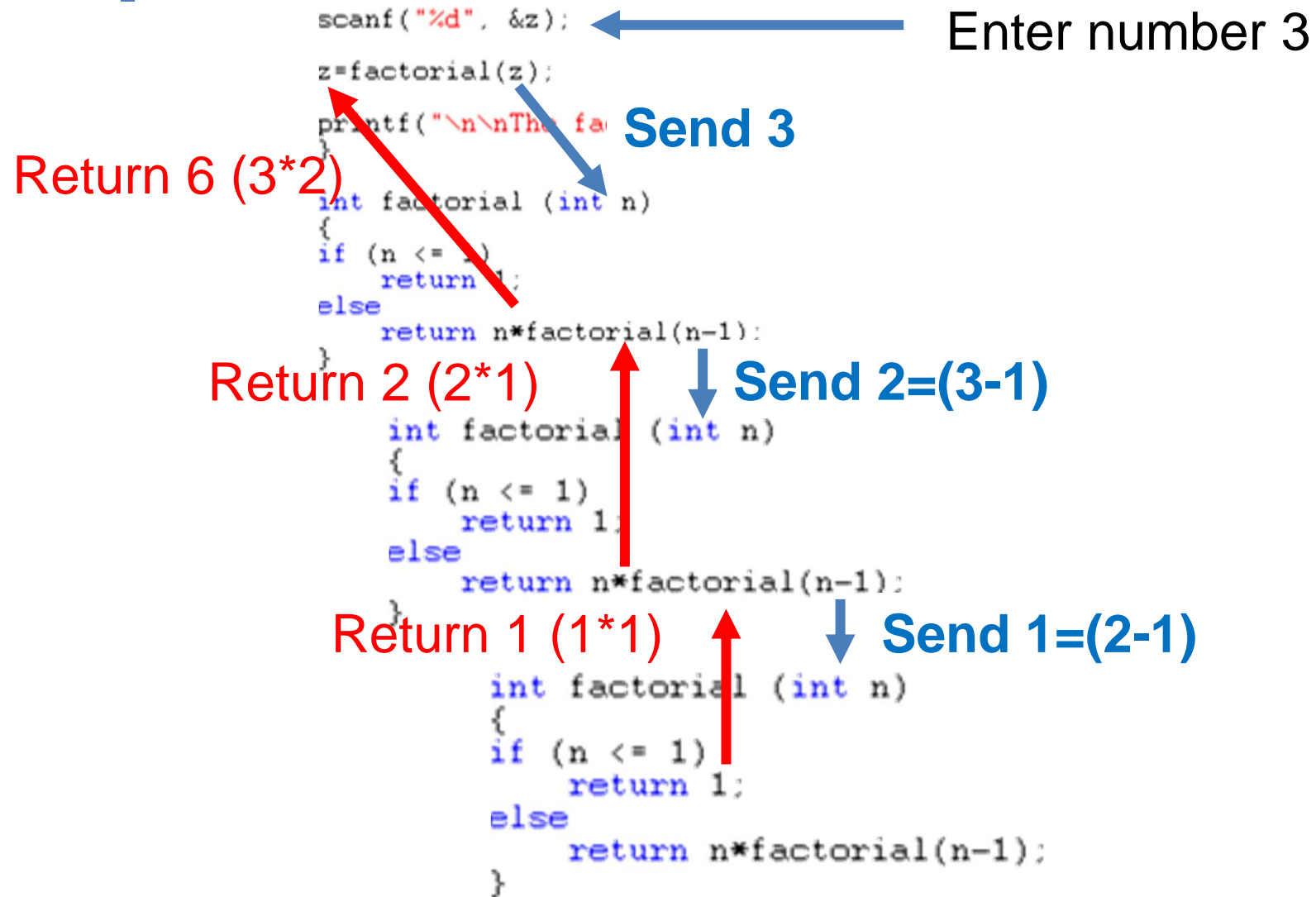
**Q. Is it possible to change the value in the function**

# Recursion

- A **recursive function** is a function that calls itself
- A classical example is the factorial function defined as  **$0! = 1$**  and  **$n! = n * (n-1)$**

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

# Recursion



# #define and the Macro

As well as using define to make replacements in your code for constants, you can also use define to produce a function:

```
#include<stdio.h>
#define MULT(a,b) (a*b)
main(){
    int w=2,x=3,y=4,z=5;
    printf("%d",MULT(w+x,y+z));
}
```

What value is printed on the screen? 19?? 😞

# #define and the Macro

As well as using define to make replacements in your code for constants, you can also use define to produce a function:

```
#include<stdio.h>
#define MULT(a,b) (a*b)
main(){
    int w=2,x=3,y=4,z=5;
    printf("%d",MULT((w+x),(y+z)));
}
```

**To make sure of your calculations always include ()**

# #define and the Macro

What about this example....Any problems?

```
#include<stdio.h>
#define max(A,B) ((A)>(B) ? (A) : (B))
main(){
    int i=1,j=2;
    printf("largest is %d\n",max(i++,j++));
    printf("i=%d and j=%d",i,j);
}
```

Think about the number of times each increment is processed

# Header Files

You should be familiar with using some of the header files  
(Note you can include as many as you like).

Dividing programs into modules/different files is sensible.  
Your code would look very complex if you had to include **all** of  
the code required for **all** functions e.g. printf() etc.

Function declarations - placed in .h files e.g. math.h

Function definitions - placed in .c files (**libraries**) e.g. math.c



# Your Own Header Files

- You can create your own libraries of your own functions.
- Your library files must be supplied for compiling.
- The compiler linker, links the code in the libraries to the function calls in your program

# Random Number Generation

- Random numbers are used in many programs, e.g. simulating noise, playing games, predictions
- C's random number generator **rand()**, generates an integer between **0** and **RAND\_MAX** (a constant defined in **<stdlib.h>**)

`i = rand();`

- The function `srand` uses a seed (e.g. the CPU clock) as the initial state for a ***pseudo random*** sequence.



# Random Number Generation

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int i;
    srand(100); /*seeds the generator*/
    for (i=1; i <= 8; i++)
    {
        printf ("%d ", 1+ (rand() % 6) );
    }
}
```

## Output:

iteration:	1	2	3	4	5	6	7	8
value:	6	5	4	1	1	5	3	5

# Random Number Generation

- Changing the seed value changes the sequence.
- The same seed will always produce the same apparently random sequence.
- Try using

```
#include<time.h>
long t;
t=time(0);
/*returns the current system clock value*/
srand(t);
```



**As always...**  
**Thank you for your attention**  
**Questions?**

**See you in the lab sessions 😊**