

COSC1101 – Programming Fundamentals

Maham Khan

Lecture – 13&14

FUNCTIONS:

A Quick Revision

Functions

- A function is a self-contained block of statements that performs coherent task of some kind.

OR

- A function groups a number of program statements into a unit and gives it a name. This unit or module can then be invoked from other parts of the program as many times as we need.

Function Syntax

```
#include <iostream.h>
```

```
void starline( );
```

```
void main ( )
```

```
{
```

```
    starline ( );
```

```
    cout<<"Hello World";
```

```
    starline ( );
```

```
    getche( );
```

```
}
```

```
void starline ( )
```

```
{
```

```
    for ( int j=0; j < 45; j++ )
```

```
        cout<<"#";
```

```
    cout<<endl;
```

```
}
```

//function declaration

//function Call

//function call

// call to library function

// starline function definition

//function body

Functions

- **Function Declaration**

- In C++, you must declare every identifier before it can be used. In the case of functions, a function's declaration must physically precede any function call.
- A function declaration announces to the compiler the name of the function, the data type of the function's return value and the data type of the parameters it uses.
- Function declarations are also called ***function prototype***.

- **Function Calls**

- A statement that transfers control to a function. In C++, this statement is the name of the function, followed by the list of arguments.
- To call a function, we use its name as a statement, with the arguments in parentheses following the name.
- A function call in a program results in the execution of the body of the called function.

Functions

- **Function Definition**

- Function Definition consists of two parts: the function **heading or declarator** and the Function **body**.
- The function body is composed of statements that make up the function, delimited by braces.
- The heading or declarator must agree with the function declaration. It must use the same function name, have the same argument types in the same order, and have the same function return type.

```
FunctionReturnType  FunctionName ( ParameterList )  
{  
    statement 1;  
    statement 2;  
    .  
    .  
    .  
}
```

Eliminating the Declaration

- Another approach of inserting a function into a program is to eliminate the function declaration and place the function definition in the listing before the first call of the function.
- Example
In our previous example the function definition of **starline()** function should come before the **main()** function. In this way we don't have to write function declaration.

Passing Arguments to Function

- An argument is a piece of data, of any data type, passed from a program to the function.
- Arguments allow a function to operate with different values.
- Arguments can be passed in two ways
 - Passing by Value
 - Passing by Reference (will be explained later)

Passing Constant as an Arguments

```
#include <iostream.h>
```

```
void starline( char, int);
```

//function declaration

```
void main ( )
```

```
{
```

```
    starline ( '*', 30 );
```

//function Call

```
    cout<<"Hello World";
```

```
    starline ( '+', 30);
```

//function call

```
    getch( );
```

```
}
```

```
// starline function definition
```

```
void starline (char ch, int n )
```

//function Declarator

```
{
```

```
    for ( int j=0; j < n; j++ )
```

//function body

```
        cout<<ch;
```

```
    cout<<endl;
```

```
}
```

Passing Variables as an Arguments

```
int calculateSum ( int, int, int );  
void main ( )  
{  
    int a, b, c, sum;  
    cout<<"Enter any three Numbers=";  
    cin>> a >> b >> c;  
    sum = calculateSum( a, b, c );  
    cout << "Sum of three number =" << sum;  
    getche( );  
}  
int calculateSum ( int x, int y, int z)  
{  
    int result;  
    result = x+y+z;  
    return result;  
}
```

Returning Values From

- When a function completes its execution, it can return a single value to the calling program.
- Usually this return value consists of an answer to the problem the function has solved.
- When a function returns a value, the data type of this value must be specified. The function declaration does this by placing the data type before the function name in the declaration and the definition.
- You should always include a function's return type in the function declaration. If you don't use a return type in the declaration, the compiler will assume that the function returns an **int** value. If function is not supposed to return anything then the type is void.

calculateSum (int , int , int)

CALLING BY REFERENCE ARGUMENTS

Call-by-value vs. Call-by-reference

- So far we looked at functions that get a copy of what the *caller* passed in.
 - This is call-by-value, as the value is what gets passed in (the value of a variable).
- We can also define functions that are passed a *reference* to a variable.
 - This is call-by-reference, the function can change a callers variables directly.

References

- A *reference* variable is an alternative name for a variable. A *shortcut*.
- A reference variable must be initialized to *reference* another variable.
- Once the reference is initialized you can treat it just like any other variable.

Reference Arguments

- Passing arguments by **reference** is a mechanism in which a reference(memory address) to the original variable, in the calling program, is passed to the function.
- When arguments are passed by **value**, the called function creates a new variable of the same type as an argument and copies the argument's value into it.

Reference Arguments

- An important advantage of passing reference arguments is that the function can access the actual variables in the calling program.
- When the function is called, the reference argument and the variable name, in the calling function, become synonyms for the same location in the memory.
- We can only use variables when we are passing arguments by reference. Whereas, we can use variables as well as constant when we are passing arguments by value.

Reference Arguments

- In passing arguments with value, implicit type conversion occurs if the matched items have different data types. Whereas, with reference arguments, the matched items must have the same data type.

Reference Variable Declarations

- To declare a reference variable you precede the variable name with a “&”:

```
int &temp;  
double &salary;  
char &ch;
```

Example

```
void abc (int&);  
void main ( )  
{  
    int temp, x;  
    abc(temp);  
    abc(x);  
    cout<<temp<<endl<<x;  
}  
void abc(int  &t)  
{  
    cout<<"enter number=";  
    cin>>t;  
}
```

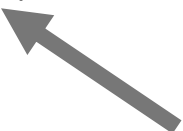
Reference Parameters

- You can declare reference parameters:

```
void add10( int &x)
{
    x = x+10;
}
```

...

```
add10(counter);
```



The parameter is a reference

Useful Reference Example

```
void swap( int &x, int &y) {  
    int tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```

CALLING BY DEFAULT ARGUMENTS

Default Arguments

- A function can be called without specifying all its arguments.
- For this purpose, function declaration must provide default values for those arguments that are not specified.

Default arguments example

```
#include <iostream.h>
void starline(char = ' # ', int = 45 ); //function declaration
void main ( )
{
    starline ( );                      //function Call
    cout<<"Hello World";
    starline ( '@' );                  //function call
    starline('$',30);
    getche( );
}
// starline function definition
void starline (char ch, int n )        //function Declarator
{
    for ( int j=0; j < n; j++ )//function body
        cout<<ch;
    cout<<endl;
}
```


EXAMPLE

CALLING MATH LIBRARY FUNCTION

rand() a Library function for the generation of random numbers

- The rand function generates an integer random ranges between
0 to 32767 (range defined in <stdlib.h>)
- We can use modulus operator (%) to limit the range as per our requirements.

Example: The following example generate 100 random numbers

Range between 0 to 32767

```
void main( )
{
    int rnum;
    for( int i=0; i<=100; i++)
    {
        rnum=rand();
        cout << rnum << \t" ";
    }
}
```

rand() a Library function for the generation of random numbers

Example: The following example generate 100 random numbers

Range between 0 to 100

```
void main( )  
{  
    int rnum;  
    for( int i=0; i<=100; i++)  
    {  
        rnum=rand( ) % 101;  
        cout << rnum << " \t";  
    }  
}
```

Random Number Generation within range

```
//generate 10 integers between 11...20
int n;
for ( int i=1; i<=10; i++){
    n= rand( )% 20 + 11;
    cout << n << "\t ";
}
```

Example : The following example generate 50 number as simulation of rolling a dice i.e. (1 to 6)

```
int n;
for (int i=1; i<=50; i++)
{
    n= rand( ) % 6 + 1;
    cout << n << "\t ";
}
```

Random Number Generator

- The `rand()` function generates the same set of random numbers every time you run the program.
- To generate a set of random numbers we can use a random generator function called `srand (unsigned integer)`
- Unsigned integer input is used as a seed. For the same seed we can generate same sequence of random number.
- library function `srand(unsigned integer);` is defined in a header file `<stdlib.h>`.
- such random number are called pseudo random numbers.

Randomizing with `srand`

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
void main()
{
    int i;
    unsigned num;
    // Please enter a different input each time
    cin >> num;
    srand (num);           // randomize the generation
    for(i=1; i<=5; i++)
        cout << 1+rand( )%6;
}
```

Random numbers without randomization

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main()
{
    int i;

    for(i=1; i<=5; i++)
        cout << 1+rand()%6;
}
```

5	3	3	5	4	2
5	3	3	5	4	2
5	3	3	5	4	2
5	3	3	5	4	2

Output for Multiple Executions

RECURSION AND RECURSIVE FUNCTIONS

Recursion and Recursive Functions

- A procedure which calls back to itself is called a recursive procedure.

similarly

- If a function calls back to it self : is called recursive function. C language allows this function calls which is not allowed by other languages like FORTRAN.
- In order to understand this concept we need to know;
- What happens when a function calls another function?
or
- What happens when a function calls back to itself ?

Function calls

fun_a()	fun_b()	fun_c()	fun_d()
{	{	{	{
100 ----- ;	200 ----- ;	300 ----- ;	400 ----- ;
110 ----- ;	210 fun_c() ;	310 ----- ;	410 ----- ;
120 ----- ;	220 ----- ;	320 ----- ;	420 ----- ;
130 fun_b() ;	230 ----- ;	330 ----- ;	430 ----- ;
140 ----- ;	240 ----- ;	340 fun_d) ;	440 ----- ; ;
150 ----- ;	250 ----- ;	350 ----- ;	450 return () ;
160 return () ;	260 return () ;	360 return () ;	
}	}	}	}

What happens when:

fun_a calls fun_b	address 140 is saved
fun_b calls fun_c	address 220 is saved
fun_c calls fun_d	address 350 is saved

350
220
140

Function calls

fun_a()	fun_b()	fun_c()	fun_d()
{	{	{	{
100 ----- ;	200 ----- ;	300 ----- ;	400 ----- ;
110 ----- ;	210 fun_c() ;	310 ----- ;	410 ----- ;
120 ----- ;	220 ----- ;	320 ----- ;	420 ----- ;
130 fun_b() ;	230 ----- ;	330 ----- ;	430 ----- ;
140 ----- ;	240 ----- ;	340 fun_d) ;	440 ----- ; ;
150 ----- ;	250 ----- ;	350 ----- ;	450 return () ;
160 return () ;	260 return () ;	360 return () ;	}
}	}	}	

What happens when:

fun_d returns	address 350 is recovered control transfers to 350
fun_c returns	address 220 is recovered control transfers to 220
fun_b returns	address 140 is recovered control transfers to 140

350
220
140

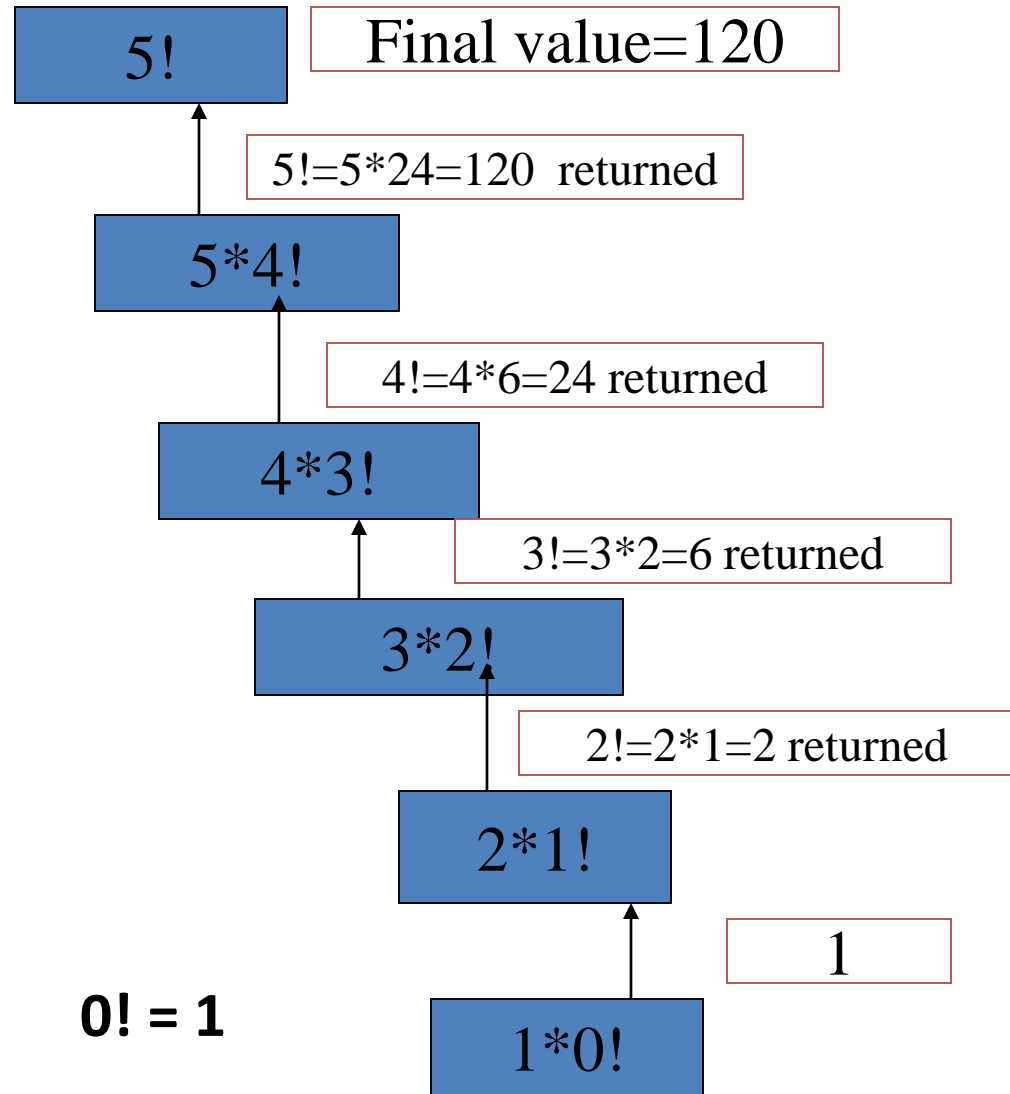
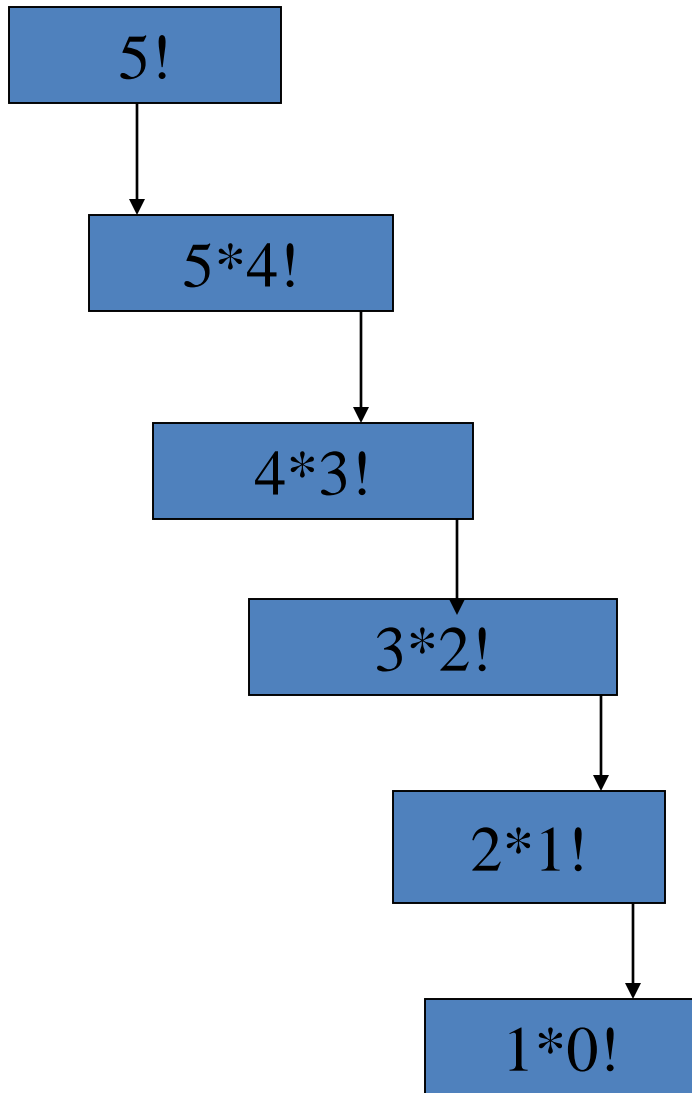
Concept Of recursion

- A recursive function calls back to it self.
- To avoid indefinite call back there exists a criteria called base criteria or so-called base-case at which function does not call back to it self.
- Thus if the function is called with a base-case, it simply returns a result.
- Ever time function calls back to itself it is closer to basic criteria/ base case. (closer to the solution)

Concept Of recursion (cont.)

- Thus the function launches (calls) a fresh copy of itself to work on the smaller problem –this is related as a Recursive-call/recursive step.
- The function keeps dividing each new sub problem into two conceptual pieces until eventually terminates after converging on the base-case.
- The function thus recognize the base-case and returns a result to the previous copy of the way up the line until original call of the function returns the final result to main.

Finding Factorial Recursively



Finding Factorial Recursively

```
//Recursive factorial Function
#include<iostream.h>
#include<iomanip.h>
unsigned long factorial(unsigned long); //prototype
int main()
{
    int num;
    cout<<"enter a positive integer:";
    cin>>num;
    cout<<"factorial="<<factorial(num);
    return 0;
}
unsigned long factorial(unsigned long n)
{
    if ( n <= 1) //the base case
        return 1;
    else
        return n * factorial (n - 1);
}
```

Finding Factorial Recursively

```
unsigned long factorial(unsigned long n)
{
    if ( n <= 1) //the base case
        return 1;
    else
        return n * factorial (n - 1);
}
```

What happens when:

factorial (5) is called

n=5, factorial (4) , is called address 1200 is saved

n=4, factorial (3) , is called address 1200 is saved

n=3, factorial (2) , is called address 1200 is saved

n=2, factorial (1) , is called address 1200 is saved

n=1, factorial (0) , is called address 1200 is saved

1200
1200
1200
1200
1200

Finding Factorial Recursively

```
unsigned long factorial(unsigned long n)
{
    if ( n <= 1) //the base case
        return 1;
    else
        return n * factorial (n - 1);
}
```

What happens when:

factorial (0) returns 1, address 1200 executes
n=1, 1X 1, returns 1, address 1200 executes
n=2, 2X 1, returns 2, address 1200 executes
n=3, 3X 2, returns 6, address 1200 executes
n=4, 4X 6, returns 24, address 1200 executes
n=5, 5X ~~24~~, returns 120 to the main function

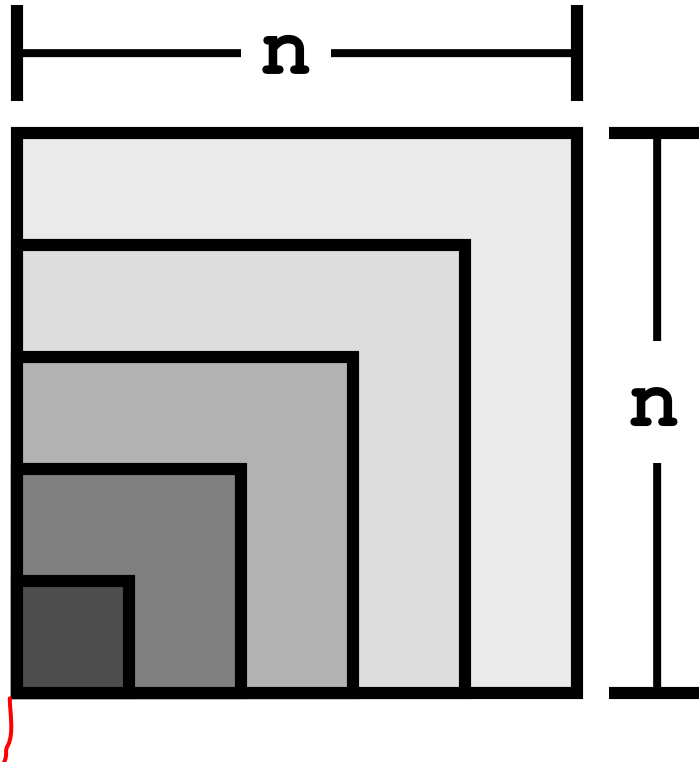
1200
1200
1200
1200
1200

Recursion Base Case

- The base case corresponds to a case in which you know the answer (the function returns the value immediately), or can easily compute the answer.
- If you don't have a base case you can't use recursion! (and you probably don't understand the problem).

Recursion is a favorite test topic

- Write a recursive C++ function that computes the area of an **$n \times n$** square.



Base case:

$n=1$ area=1

Recursive Step:

$\text{area} = n + n - 1 + \text{area}(n-1)$

Recursive area function

```
int area( int n) {  
    if (n == 1)  
        return(1) ;  
    else  
        return( n + n - 1 + area(n-1) ) ;  
}
```

EXERCISES

RECURSIVE FUNCTIONS

Example of Recursion

```
aint isprime(int, int = 2); //prototype
int main() {
    int num;
    cout<<" \n enter a positive integer:\n";
    cin>>num;
    if (isprime(num))
        cout << num << " is a prime number\n";
    else
        cout << "Not a Prime Number\n";
    system ("pause");
    return 0;
}
int isprime(int p, int i) {
    if (i==p)                                //or better if (i*i>p) return 1;
        return 1;
    if (p%i == 0)
        return 0;
    else
        return isprime (p, i+1);
}
```