Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

I2204
Imperative Programming

Dr Siba Haidar

About this course

- 3 sessions per week

Day	What	Time	Platform
Monday	Lecture (recorded offline)	9:50 – 11:30	YouTube
Wednesday	LAB	9:50 – 11:30	(Hacker Rank |Moodle) & Microsoft Teams
Friday	Exercises	9:50 – 11:30	Microsoft Teams

- prerequisites: I1101 (old INFO203) → Imperative Programming I
- 50 hours
- resources to be available online after each lesson

Course Objectives

- The purpose of this module is to
  - deepen the study of the imperative programming
  - through the use of advanced aspects of the imperative language seen in I1101

- The student must be able to
  - implement the concepts covered in this course
  - to create an application solving a complex problem as modules

Course Outline

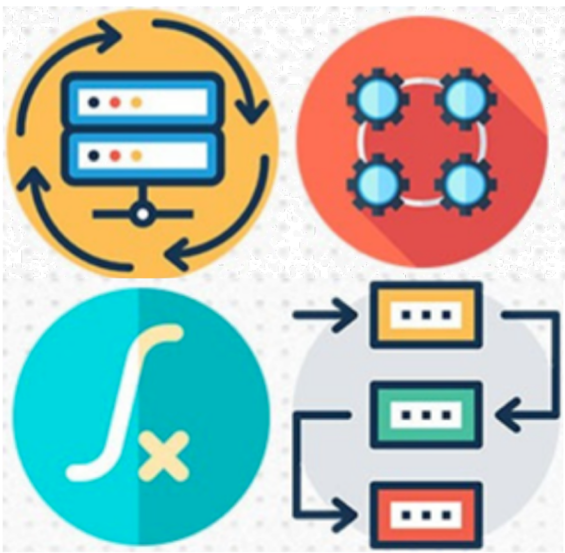
- recursive functions
- structures
- pointers & arrays
- linked lists
- input / outputs

Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

Recursion

Chapter 1

Recursion



1. The memory state and function calls
2. Recursive functions, their types and rules

Exercise 1

- Write a C program which reads an integer number, calculates its double and displays it on the screen
- Draw the memory state

Exercise 2

- Write a program which does the same as in exercise 1 but this time using functions
- the program calls
  - a function "readNum" to read a number and return it
  - a function "doubleIt" to calculate the double of the number
  - a third function "display" to display the result
- Draw the memory state

Exercise 3

- Write a program which
  - calls a function "read10Nums" to read 10 integers
  - calls another function "avg" to calculate and return the average of the integers
  - the function "display" to display the result
- Draw the memory state

Recall

- how to give functions a type?
- what are the arguments used for?
- can we call any function in C from within any other function?
- can we define any function in C within any other function?
- what is the difference of parameter and arguments?

Parameter vs. Argument

- parameter
  - refers to any declaration within the parentheses following the function name
  - in a function declaration or definition
  - ex:
    - int max(int a, int b);
- argument
  - refers to any expression within the parentheses
  - of a function call
  - ex:
    - int m=max(3,x);

Definition & Declaration

- defining a function is where you actually provide a definition
  - what the function actually does
  - between { }
    - int addTwo(int a, int b) { return a + b; }
- declaring a function is simply telling the compiler about the function
    - int addTwo(int a, int b);
    - you can also write
    - int addTwo(int, int);

Function call

- every function in C may be called from any other or itself
- each invocation of a function causes a new allocation of the variables declared inside it
- declarations had something missing
  - keyword auto → ‘automatically allocated’

Example

```c
int myfunction (int a, float b){
        int r;
        r = a + (int) b * 2;
        return r;
}
```

function definition

```c
e = myfunction (2, 3.4);
```

function call

The Keyword auto

- storage for auto variables →
  - automatically allocated on function entry
  - automatically freed on function return

```
int main() {
    auto int var_name;
    //...
    return 0;
}
```

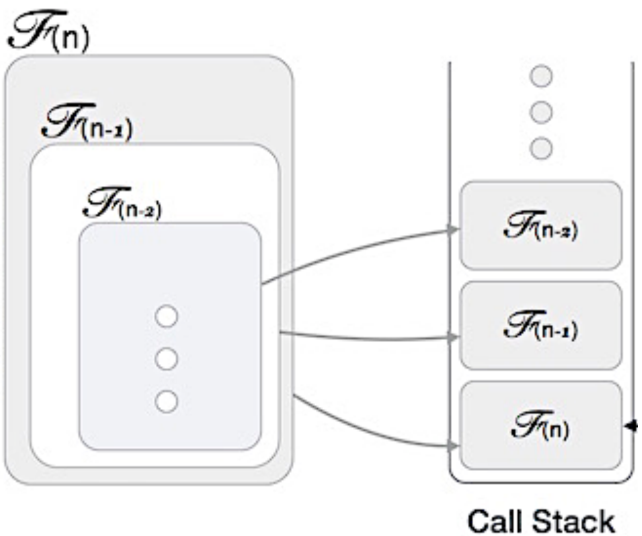
Exercise 4

- Write a program which does calls
  - the function "readNum" to read a number and return it
  - the function "doubleIt" to calculate the double of the number
  - then again the function "readNum" to read another number and return it
  - and the function "doubleIt" to calculate the double of that other number
  - a third function "display" to display the sum of the results
- Draw the memory state

Test functions

- every time we write a function "anyFunction"
  - we must write another void function to test it
  - we call it "anyFunctionTest"
    - same name with Test suffix
  - the test function must not read inputs from the keyboard
    - for not to waste time
    - it provides static test values
  - should try to cover all test cases

Recursion



1. The memory state and function calls
2. Recursive functions, their types and rules

Recursion

- what is recursion?
  - when one function calls ITSELF directly or indirectly.
- why learn recursion?
  - new mode of thinking
  - powerful programming tool
  - divide-and-conquer paradigm
- many computations are naturally self-referential
  - a directory contains files and other directories
  - Euclid's gcd algorithm
  - quicksort algorithm
  - linked data structures

Recursive Function

- a recursive function definition has
  - one or more base cases,
    - input(s) for which the function produces a result trivially (without recurring), and
  - one or more recursive cases,
    - input(s) for which the program recurs (calls itself)

Example: Factorial

- factorial function can be defined recursively by the equations
  - $0! = 1$ and,
  - for all $n > 0$, $n! = n \times (n - 1)!$
- neither equation by itself constitutes a complete definition;
  - the first is the base case
  - the second is the recursive case

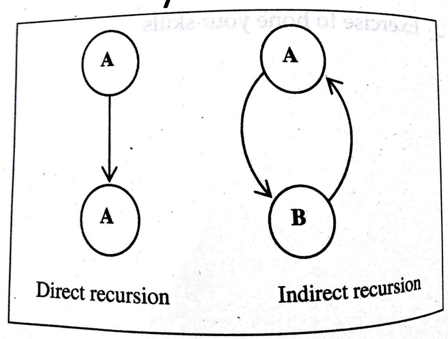
- because the base case breaks the chain of recursion, it is sometimes also called the "terminating case"

Example: Factorial

```c
#include<stdio.h>
int factorial(int n) {
    if (n<0)
        return -1; //we do not treat <0 numbers
    else
        if(n==0)
            return 1;
    return factorial(n-1) * n;
}
void factorialTest() {
    printf("Factorial of %d = %d",-1, factorial(-1));
    printf("Factorial of %d = %d", 0, factorial(0) );
    printf("Factorial of %d = %d", 3, factorial(3) );
}
int main(){
    factorialTest();
    return 0;
}
```

Types of Recursion

- Direct Recursion
  - A function is said to be direct recursive if it calls itself directly.




```
int fibo(int n) {
    if ( n == 1 || n == 2)
        return 1;
    return ( fibo(n-1) + fibo(n-2) );
}
```

- Indirect Recursion
  - A function is said to be indirect recursive if it calls another function and this new function calls the first calling function again.

```
int func2(int);

int func1(int n) {
    if ( n <= 1)
        return 1;
    return func2(n);
}

int func2(int n) {
    return func1(n);
}
```

Quick Sort Algorithm

```
/* Double-Click To Select Code */

  function quicksort('array')
      if length('array') ≤ 1
         return 'array'  // an array of zero or one elements is already sorted
      select and remove a pivot value 'pivot' from 'array'
      create empty lists 'less' and 'greater'
      for each 'x' in 'array'
         if 'x' ≤ 'pivot' then append 'x' to 'less'
         else append 'x' to 'greater'
      return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
// two recursive calls
```

Example 1: https://youtu.be/tIYMCYooo3c

Example 2: https://youtu.be/cnzIChso3cc

Greatest Common Divisor (gcd)

- find largest integer d that evenly divides into p and q
- example
  - suppose p = 32 and q = 24
  - integers that evenly divide both p and q: 1, 2, 4, 8
  - ➔ d = 8 (the largest)
- how would you compute gcd?

Greatest Common Divisor (gcd)

- find largest integer d that evenly divides into p and q

$$gcd(p, q) = \begin{cases} p & \text{if } q = 0 \\ gcd(q, p \% q) & \text{otherwise} \end{cases}$$

⬅ base case

⬅ reduction step, converges to base case

```
gcd(4032, 1272) = gcd(1272, 216)
                = gcd(216, 192)
                = gcd(192, 24)
                = gcd(24, 0)
                = 24
```

$4032 = 2^6 \times 3^2 \times 7^1$

$1272 = 2^3 \times 3^1 \times 53^1$

$gcd = 2^3 \times 3^1 = 24$

Greatest Common Divisor (gcd)

- find largest integer d that evenly divides into p and q

$$\gcd(p, q) = \begin{cases} p & \text{if } q = 0 \\ \gcd(q, p \% q) & \text{otherwise} \end{cases}$$

⬅ base case

⬅ reduction step, converges to base case

```
int gcd (int p, int q){
    if (q == 0)
        return p;                    ⬅ base case
    return gcd(q, p % q);            ⬅ reduction step
}
```

Tracing Recursive Functions

- "winding" part
  - recursion heads to base case
  - example: a() calls b(), and b() calls c(), and c() calls d()

- "unwinding" part
  - returns back to original call
  - example: d() done, it goes back to c(), … to b(), … to a()

Exercise 5

- Write a program in C to calculate the sum of numbers from 1 to n using recursion.
  - Test Data :
    - Input the last number of the range starting from 1 : 5
  - Expected Output :
    - The sum of numbers from 1 to 5 : 15

Exercise 6

- Write a program in C to count the digits of a given number using recursion.

- Test Data :
  - Input a number : 50
- Expected Output :
  - The number of digits in the number is : 2

Why is this wrong?

```
int noOfDigits(int n1) {
    static int ctr=0;
    if(n1!=0) {
        ctr++;
        noOfDigits(n1/10);
    }
    return ctr;
}
```

Tail Recursion

- tail-recursive function
  - no additional work after recursive call
  - except return
  - *often require an additional parameter*

```
int tailRec(int x, int y){
    if(...){
        ...
    }
    else{
        ...
        return tailRec(...,...);
    }
}
```

- non tail-recursive functions
  - after the recursive call there is still work to do

```
int nonTailRec(int x, int y){
    if(...){
        ...
    }
    else{
        ...
        return nonTailRec(...,...) + 3;
    }
}
```

Tail Recursion

- The goal of tail recursion in its simplest form is to return the answer that we have accumulated throughout all of the function calls in the last frame.

Exercise 7: Recursive Print

- is print a tail-recursive function?
- output?

```c
#include <stdio.h>
void print(int n){
    if(n < 1)
        return;
    print(n -1);
    printf("%d\n", n);

}
void printTest(){
    print(5);
}
int main(){
    printTest();
    return 0;
}
```

```
Print(5)
  Print(4)
    Print(3)
      Print(2)
        Print(1)
          Print(0) <-- base case here
        Print(1) <-- prints 1
      Print(2) <-- prints 2
    Print(3) <-- prints 3
  Print(4) <-- prints 4
Print(5) <-- prints 5
```

Exercise 8

- write a tail-recursive function which produces the inverse output as the previous exercise
- when called with parameter n=5

5

4

3

2

1

Exercise 9

- write a recursive function which prints on the screen a triangle of stars pointing up
- example: triangle(9)

```
*
***
*****
*******
*********
```

```
#include <stdio.h>
void triangle(int n){
    int i = 0;
    if (n <=0)
        return;
    triangle(n-2);                    rec call
    for(i=0;i<n;i++)
        printf("*");                  non-tail
    printf("\n");
}
void triangleTest(){
    triangle(9);
}
int main(){
    triangleTest();
    return 0;
}
```

Exercise 10

- write a recursive function which prints on the screen a triangle of stars pointing down
- example: triangle(9)

```
*********
*******
*****
***
*
```

```
#include <stdio.h>
void triangle(int n){
    int i = 0;
    if (n <=0)
        return;
    for(i=0;i<n;i++)
        printf("*");
    printf("\n");
    triangle(n-2);
}
void triangleTest(){
    triangle(9);
}
int main(){
    triangleTest();
    return 0;
}
```

rec call *tail*

Exercise 11

- write a tail recursive factorial function

- you can use a helper function
  - a helper function???

What are Helper Functions?

- Helper functions are useful when you want to extend the amount of parameters that a certain function takes in.
- Helper functions are generally used to make our lives easier.
- This occurs most often when working with recursion, especially if you want your function to be tail recursive.

Let’s look again at Factorial

```
#include<stdio.h>
int factorial(int n) {
    if(n==0)
        return 1;
    return factorial(n-1) * n  ;
}
void factorialTest() {
    int num=3,f;
    f=factorial(num);
    printf("Fact %d = %d",num,f);
}
int main(){
    factorialTest();
    return 0;
}
```

- We know that we can’t do this while only taking in a single parameter, n, so we look to create a helper function.

Exercise 11

```c
#include<stdio.h>
int factorialHelper(int, int);


int factorial(int n){
    if (n < 0)
        return -1;
    return factorialHelper(n, 1);
}
int factorialHelper(int n, int p) {
    if( n==0 )
        return p;
    p *= n;
    return factorialHelper (n-1, p);
}
```

```c
void factorialTest() {
    int num=3,f;
    f=factorial(num);
    printf("Fact %d = %d",num, f);
}
int main(){
    factorialTest();
    return 0;
}
```

Exercise 12

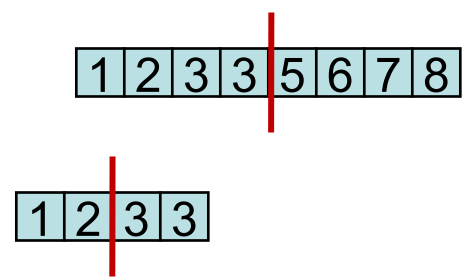
- Write a program in C to print the array elements using recursion.

Exercise 13

- Write a program in C to check whether a given string is a palindrome or not.
  - Input a word to check for palindrome : mom
  - Expected Output :
    - The entered word is a palindrome.

Exercise 14

- write a tail recursive function
  - to apply binary search inside sorted arrays
  - find whether a given number n is inside the array
    - no → return -1
    - yes → return its first occurrence index
- example
  - find if 3 is there



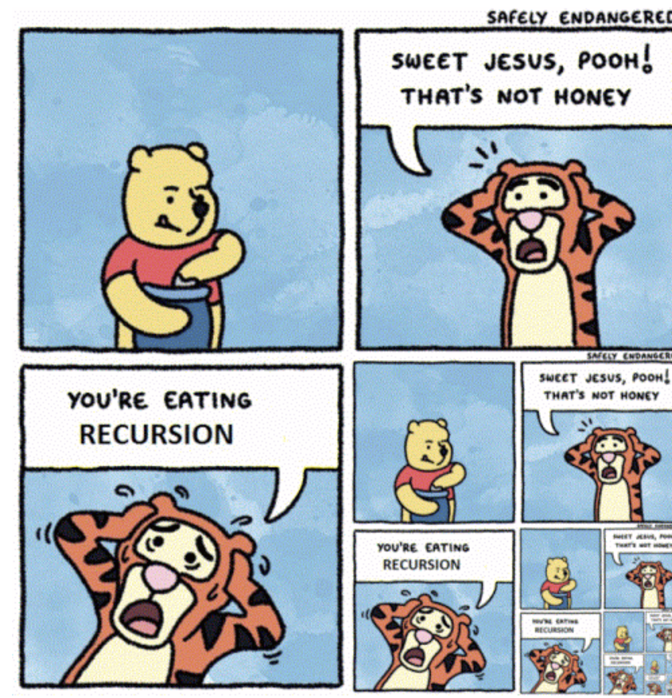
Rule of Thumb

- Tail-recursive functions are faster if they don't need to reverse the result before returning it.

THE EFFECTIVENESS OF RECURSION

Possible Pitfalls With Recursion

- recursion can potentially consume more memory than an equivalent iterative solution
  - because the latter can be optimized to take up only the memory it strictly needs
  - but recursion saves all local variables on the stack
  - thus taking up a bit more than strictly needed
- recursion can take a long time if it needs to repeatedly recompute intermediate results

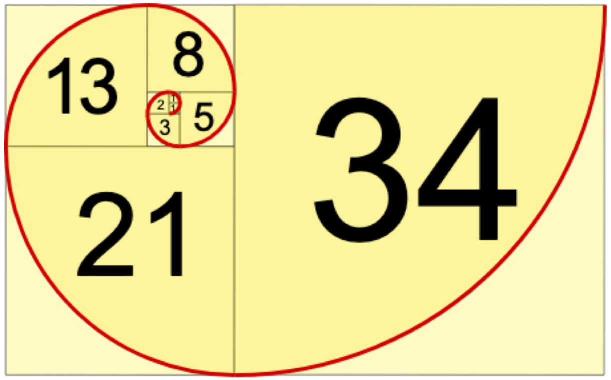


Example: Fibonacci Numbers

- infinite serie
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . .
- a natural for recursion

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

L. P. Fibonacci
(1170 - 1250)



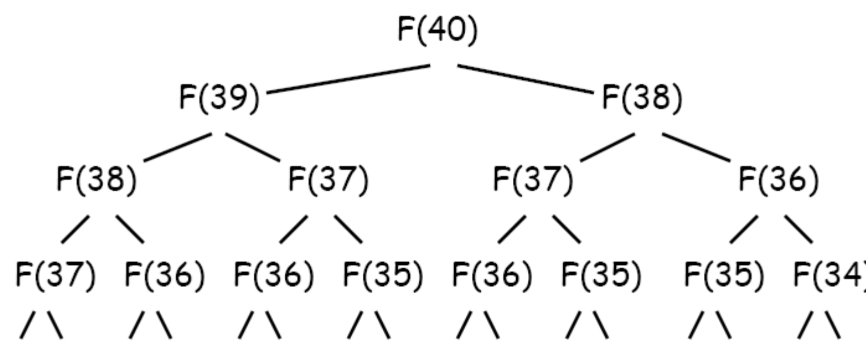
Example: Fibonacci Numbers

- What about this solution:

```
int F(int n){
    if (n == 0 || n == 1)
        return n;
    return F(n-1) + F(n-2);
}
```

- Spectacularly inefficient Fibonacci!
  - why?
- takes really long time to compute F(40)= ?
  - F(39) is computed once
  - F(38) is computed twice
  - F(37) is computed 3 times
  - F(36) is computed 5 times
  - F(35) is computed 8 times
  - ...
  - F(0) is computed 165,580,141 times.

- → Overlapping cases!



Exercise 15

- Can you write a better tail-recursive Fibonacci function?

What is Recursion Good for?

- can reduce time complexity
  - if you ***memoize*** the result to avoid overlapping (if any)
- adds clarity and reduces the time needed to write and debug code
  - if input is **small**
- **better at tree traversal**