BORG AL ARAB TECHNOLOGICAL UNIVERSITY

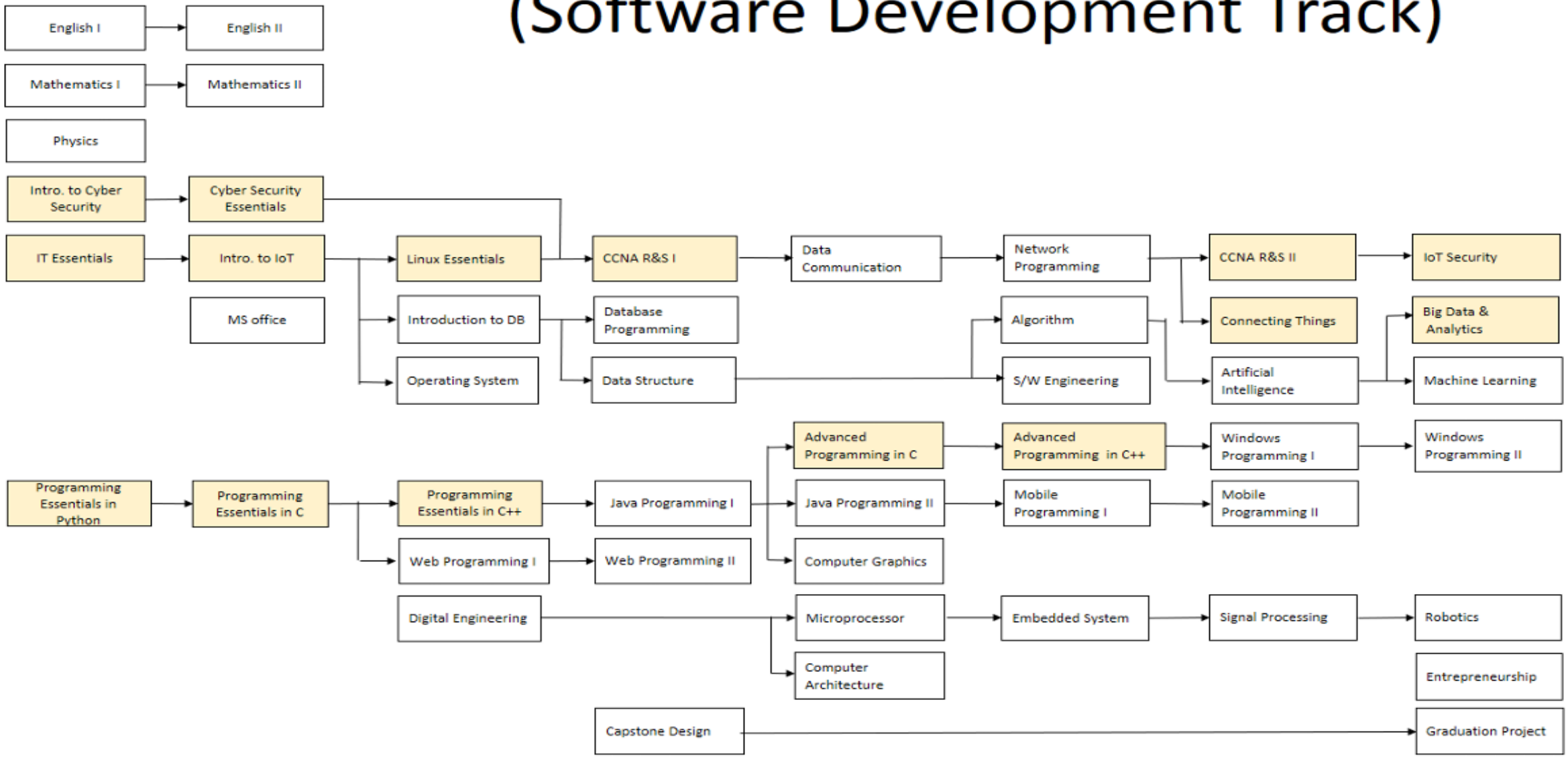
CODE: ITS501

ADVANCED PROGRAMMING IN C

COURSE INFORMATION

- Lecturer: Prof. Dina Abdelhafiz
- Credit hours (3)
- Requirements & Grading (Total 150 marks)
  - Class work and attendance (40 marks)
  - Midterm exam (35 marks)
  - Final Exam during finals week (75 marks)

Course Hierarchy (Software Development Track)

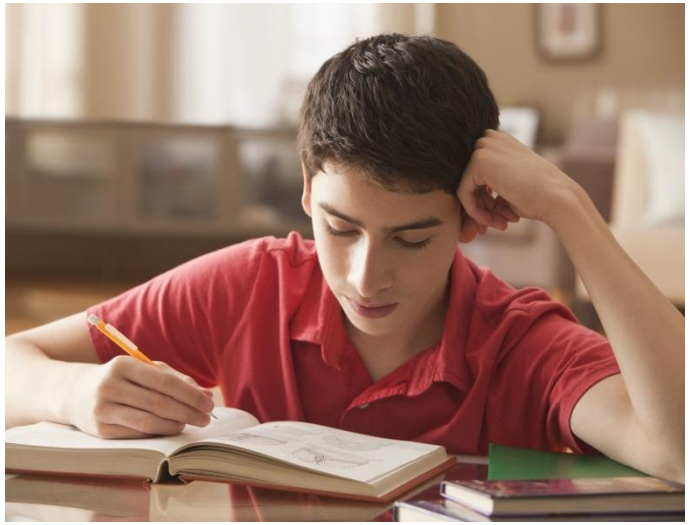


ABOUT EXAMS

- All material included in the lectures uploaded to LMS.
- Notes taken from lecturer at lecture time.
- All material included in the section by your Teaching assistant (TA).

WHAT YOU NEED TO PASS THE EXAMS

Just attend the lectures, take notes and study



LECTURE 1

Introduction To Functions In C Programming

WHAT ARE FUNCTIONS?

- A function is a block of code which only runs when it is called.
- You can pass data, known as inputs or parameters, into a function.
- Functions are used to perform certain actions.
- they are important for reusing code: Define the code once, and use it many times.
- Functions can be called from anywhere in a program, allowing for better organization.

WHY USE FUNCTIONS?

- Write once, use multiple times.
- Break down complex problems into manageable pieces.
- Easier Debugging: Isolate and fix issues within specific functions.
- Organized Code Structure: Enhances readability and maintainability.

TYPES OF FUNCTIONS

- **Library Functions – Predefined** (main, printf, scanf, sqrt)

- **User-defined Functions or declared functions**– Written by programmer

PREDEFINED FUNCTIONS

So it turns out you already know what a function is.

- For example, main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen:

```c
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

BASIC FUNCTION SYNTAX (DECLARED FUNCTIONS)

- To **create** (***declare***) your own function, specify the name of the function, followed by parentheses () and curly brackets {}

```
return_type function_name(parameters) {
    // function body
}
```

- return_type: Specifies the type of value the function returns (e.g., int, float, void if no value is returned).
- function_name: The name of the function.
- parameters: Inputs that the function can accept to perform operations (optional).
- function body {}: Contains the statements that define what the function does.

RULES FOR NAMING A FUNCTION

- A function name must begin with a letter or underscore (_).

 addNumbers  2addNumbers

- Only letters, digits, and underscores are allowed.

print_message print-message (hyphens not allowed)

- Function names are case-sensitive.

sum() and **S**um() are different.

- Function names cannot be a reserved keyword in C.

 int()  return() are invalid.

- No fixed maximum length in standard C, but make it simple, related to its task and readable.

RULES FOR NAMING PARAMETERS IN C

- Same rules as variable naming: must start with a letter or _, followed by letters, digits, or _


int age


int 1age

- Unique within the parameter list.


int add(int x, int x)` is invalid.

- Cannot use C keywords as parameter names.


int add(int int)

- Parameters are local variables inside the function (**they exist only during function execution**).

BEST PRACTICES (WHAT TO DO AND WHAT TO AVOID!)

Good Practices for Function & Parameter Naming

Descriptive names

 Good: calculateArea(), findMax(int num1, int num2)  Bad: func1(), doStuff()

Use verbs for function names (functions perform actions)

 printMessage(), getAverage(), sortArray()

Use nouns for parameters (they represent data)

 int radius, float temperature, char name[]

Consistency in style

 Stick to **camelCase** (calculateSum) or **snake_case** (calculate_sum) but don't mix.

BEST PRACTICES (WHAT TO DO AND WHAT TO AVOID!)

Good Practices for Function & Parameter Naming

- Avoid naming global variables with same names as parameters.
- Makes functions reusable.
- Keep functions small and focused (One function = one purpose).
- Comment functions: Briefly describe what the function does, input, and output.

CALL A FUNCTION

- Declared functions **are not executed** immediately.
- They are "saved for later use", and will be executed when they are called.
- To call a function, write the function's name followed by two parentheses () and a semicolon ;

CALL A FUNCTION

```c
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

In the following example, myFunction() is used to print a text (the action), when it is called:

A FUNCTION CAN BE CALLED MULTIPLE TIMES

- A function can be called **multiple times**:

```c
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

EXAMPLEOF DEFINNING AND CALLING A FUNCTION PRINT A MESSAGE

Code Example:

```c
#include <stdio.h>

void printMessage() {    // Function definition
    printf("Hello, World!\n");
}

int main() {
    printMessage();     // Function call
    return 0;
}

//Hello, World!
```

- void printMessage(){}

Defines a function that does not return any value and takes no parameters.

- printf("Hello, World!\n");

Executes within the function to print the message.

- printMessage();

Calls the printMessage function from main, triggering the message to be printed.

EXAMPLE 2

```c
#include <stdio.h>

// Create a function
void calculateSum() {
    int x = 5;
    int y = 10;
    int sum = x + y;
    printf("The sum of x + y is: %d", sum);
}

int main() {
    calculateSum();   // call the function
    return 0;
}
// The sum of x + y is: 15
```

- The real power of a function is when we pass "parameters" to it.
- This allows the function to calculate the sum of any numbers, instead of being limited to the fixed values 5 and 10.

PASSING PARAMETERS TO FUNCTIONS

Parameters are specified after the function name, inside the parentheses.

You can add as many parameters as you want, just separate them with a comma ,

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

PASSING MULTIPLE ARGUMENTS

- Functions can accept multiple parameters, allowing them to process more data.

```c
void calculateSum(int x, int y) {
  int sum = x + y;
  printf("The sum of %d + %d is: %d\n", x, y, sum);
}

int main() {
  calculateSum(5, 3);
  calculateSum(8, 2);
  calculateSum(15, 15);
  return 0;
}
```

CALLING OF FUNCTION

If we consider the **"Calculate the Sum of Numbers"** example one more time, we can use return instead and store the results in different variables. This will make the program even more flexible and easier to control:

```c
int calculateSum(int x, int y) {
  return x + y;
}

int main() {
  int result1 = calculateSum(5, 3);
  int result2 = calculateSum(8, 2);
  int result3 = calculateSum(15, 15);

  printf("Result1 is: %d\n", result1);
  printf("Result2 is: %d\n", result2);
  printf("Result3 is: %d\n", result3);

  return 0;
}
```

PASSING PARAMETERS TO FUNCTIONS

- Allow functions to accept input values, making them more flexible and reusable.
- Parameters act as variables within the function.

```c
void myFunction(char name[], int age) {
  printf("Hello %s. You are %d years old.\n", name, age);
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Hello Liam. You are 3 years old.
// Hello Jenny. You are 14 years old.
// Hello Anja. You are 30 years old.
```

Notes on Parameters

- When a **parameter** is passed to the function, it is called an **argument**.
- Note that when you are working with multiple parameters, the function call must **have the same number of arguments** as there are parameters, and the arguments must be passed in the same order.

RETURNING VALUES FROM FUNCTIONS

- Return
  - Used to send a value back to the function caller.
  - Terminates the function execution.

```c
int myFunction(int x) {
  return 5 + x;
}

int main() {
  printf("Result is: %d", myFunction(3));
  return 0;
}

// Outputs 8 (5 + 3)
```

PASSING VALUES

- In C, arguments are passed by value, meaning a copy of each argument is made.

- Changes made to parameters inside the function do not affect the original arguments.

PASSING BY VALUE

- void increment(int num): Function that attempts to increment num.
- num = num + 1;: Increments the local copy of num.
- printf("Inside function: num = %d\n", num);: Shows num as 6 inside the function.
- int a = 5;: Original variable.increment(a);: Passes a copy of a to the function.
- printf("After function call: a = %d\n", a);: Shows a remains 5 after the function call.

```c
void increment(int num) { // Pass by value
    num = num + 1;
    printf("Inside function: num = %d\n", num);
}

int main() {
    int a = 5;
    printf("Before function call: a = %d\n", a);
    increment(a); // Function call
    printf("After function call: a = %d\n", a);
    return 0;
}
```

NAMING VARIABLES

- If you operate with the same variable name inside and outside of a function, C will treat them as two separate variables;
- One available in the global scope (outside the function) and one available in the local scope (inside the function):

NAMING VARIABLES

- The function will print the local x, and then the code will print the global x:

```c
#include <stdio.h>

// Global variable x
int x = 5;

void myFunction() {
  // Local variable with the same name as the global variable
  int x = 22;
  printf("%d\n", x); // Refers to the local variable x
}

int main() {
  myFunction();

  printf("%d\n", x); // Refers to the global variable x
  return 0;
}
```

FUNCTION DECLARATION

- What is a Function Declaration?
  - Also known as a **function prototype**.
  - A function prototype is a declaration of a function that **specifies the return type and parameters but does not include the body**.
  - Allows the compiler to ensure that functions are called correctly with the right arguments.

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

FUNCTION DECLARATION(PROTOTYPE)

```c
int add(int a, int b); // Function declaration

int main() {
    int sum = add(5, 3); // Function call
    printf("Sum: %d\n", sum);
    return 0;
}

int add(int a, int b) { // Function definition
    return a + b;
}
```

int add(int a, int b);

Declares a function named add that takes two int parameters and returns an int.

int sum = add(5, 3);:

Calls the add function with arguments 5 and 3.

return a + b;

The function adds the two parameters and returns the result.

COMMON MISTAKES IN FUNCTIONS

- **Forgetting Return Statements**: Functions with a non-void return type must return a value.
- **Mismatched Data Types**: Ensure the return type and parameter types match their declarations.

Happy Coding !