UNIT 5. FUNCTIONS

Programming

Year 2017-2018

Grade in Industrial Technology Engineering

Contents

Programming Paradigms

- What is a programming paradigm?
  - Basic criteria that rule the design of a programming language
  - a style of building the structure and elements of computer programs

- Some paradigms
  - Structured programming
  - Modular programming
  - Object-oriented programming

Programming Paradigms

- Whe need programming techniques that help us develop good programs

- What is a **good program**?
  - **Correct** Produces the required results
  - **Easy to debug**: Designed in a way that facilitates error location and correction
  - **Easy to extend** : The program facilitates adding new functionalities
  - **Readable :** easily understood by any other programmer

Modular programming

- Modular programming:
  - Based on the decomposition of a problem in simpler problems (modules) that can be analysed, programmed, debugged and tested independently

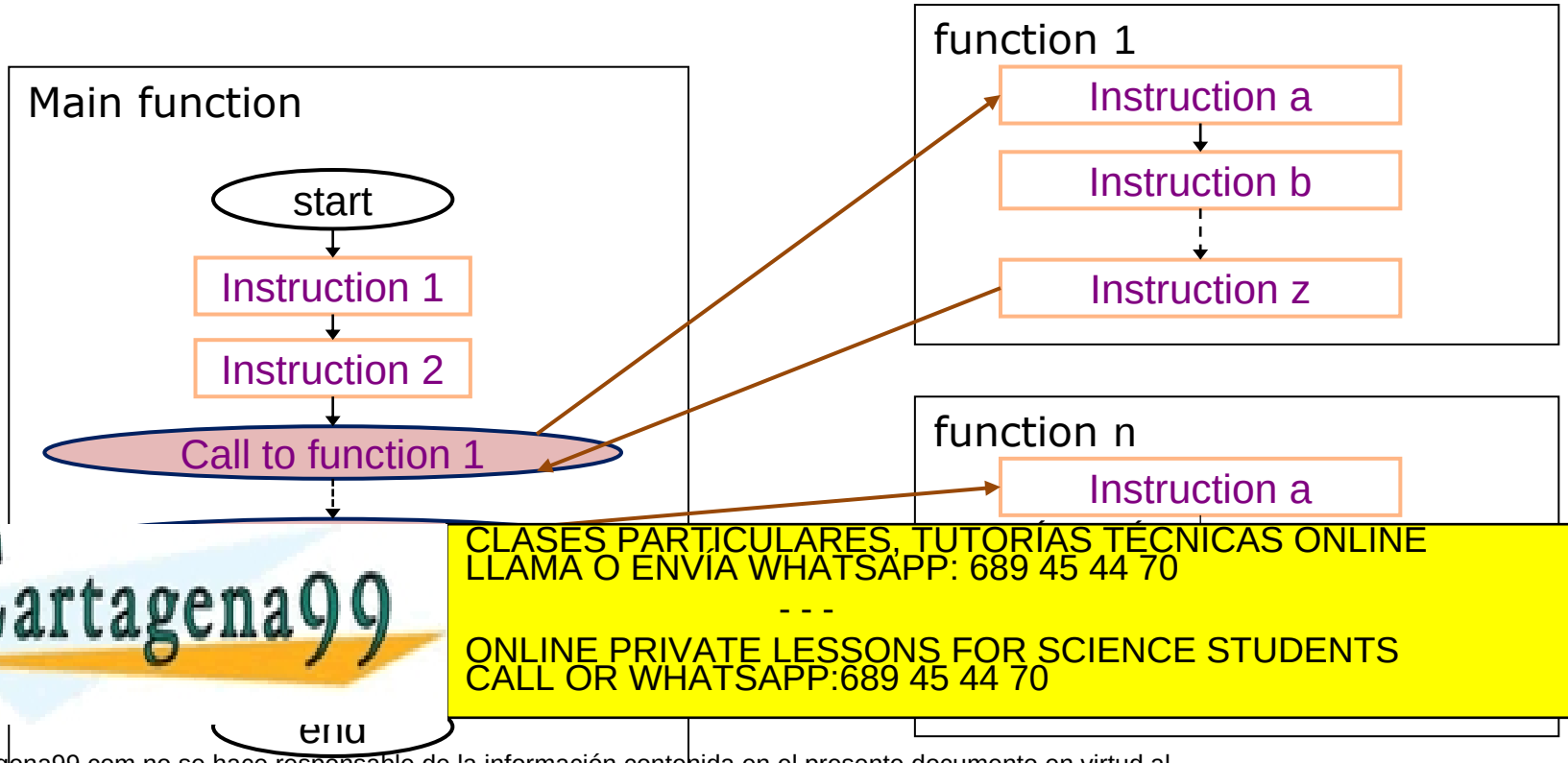
- A module is:
  - A set of instructions that perform a specific task or provide results, that can be called from the main function or from other modules
  - Module, subprogram, functions – synonyms
    - In C, we call them functions

Advantages of modular programming

- More structured and easier to read programs
  - Shorter and simpler programs, thanks to the modularity
- Subprograms are independent
  - They can be **created, compiled and tested** independently, and therefore different people can work together in a large software project
  - A subprogram can be **modified** without having to change the rest of the program, nor testing it again
- **Subprograms** are **reusable.**

Modular programming

- A program comprises:
  - **Main function,** containing general program logic and calls to subprograms
  - **Subprograms (functions)** : independent modules to solve specific problems



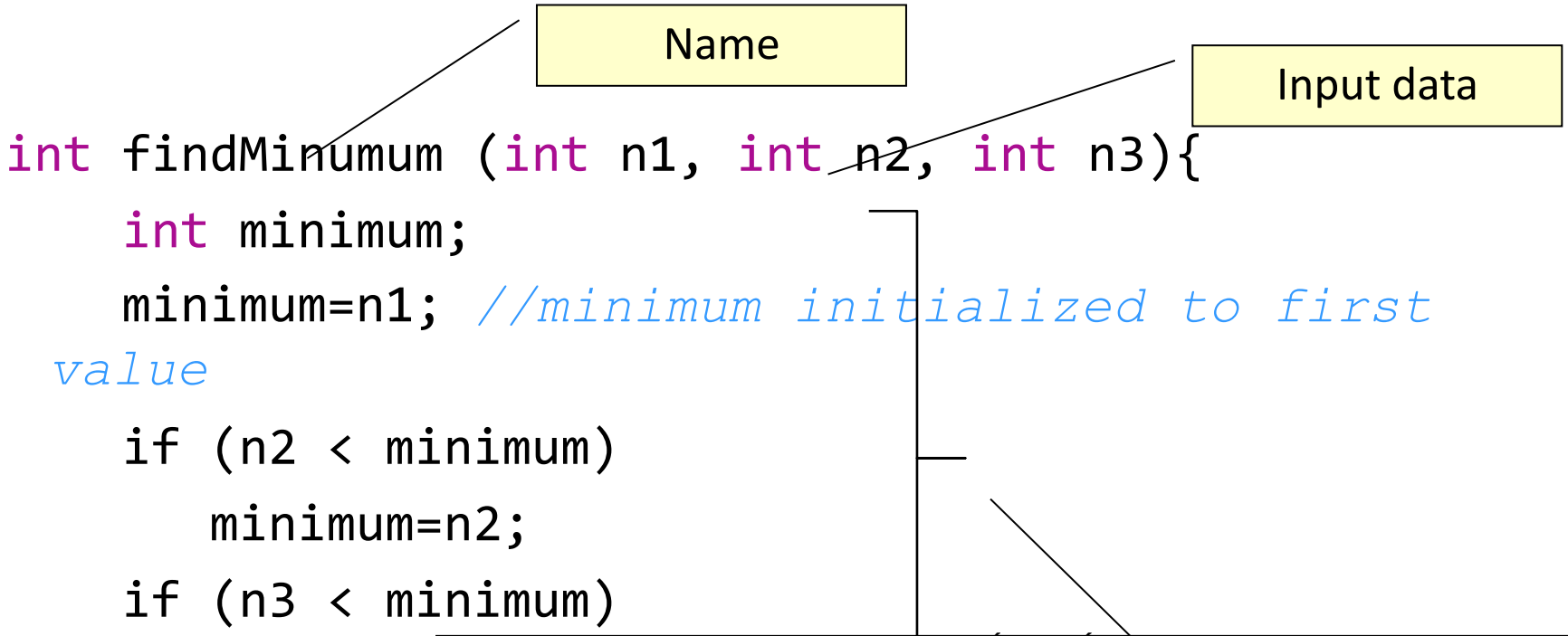
Modular programming

- **Making a good division of a program in functions is a key aspect in software development**

5.2. FUNCTION DECLARATION AND DEFINITION

Example of a function

- A function is a block of code that solves a particular problem



Return value of a function

- A function performs a set of tasks and returns a **result**
  - Also named **return** value
- When declaring a function we have to define the **data type** of that result
  - It can be int, float, double, char
  - It's also possible that a function doesn't have a return value
    - This is specified using the keyword **void**

```
int findMinumum (int n1, int n2, int n3){
    int minimum;
    minimum=n1;
    if (n2 < minimum)
        minimum=n2;
}
```

Return value

Parameters and arguments

- Parameters and arguments
  - Parameters are the symbolic name for data that goes into a function
  - Arguments are the actual data we pass into the functions parameters
  - Each parameter is of a specific data type
  - There can be one, more or none parameters
    - If there are no parameters, we use **void** keyword to tell the computer this

Function definition: header

- Function definition has two parts: header and body
- Header
  - First line of the function
  - Contains all essential information regarding the function
- Examples

```
int findMinimum (int n1, int n2, int n3)
```

```
float add(float a, float b)
```

Function definition: body

- Body
  - Block of code that is executed in every call to the function
  - They perform the function's task

- example:

```
{
    int minimum;
       minimum=n1;
       if (n2 < minimum)
          minimum=n2;
       if (n3 < minimum)
          minimum=n3;
}
```

Return value of a function

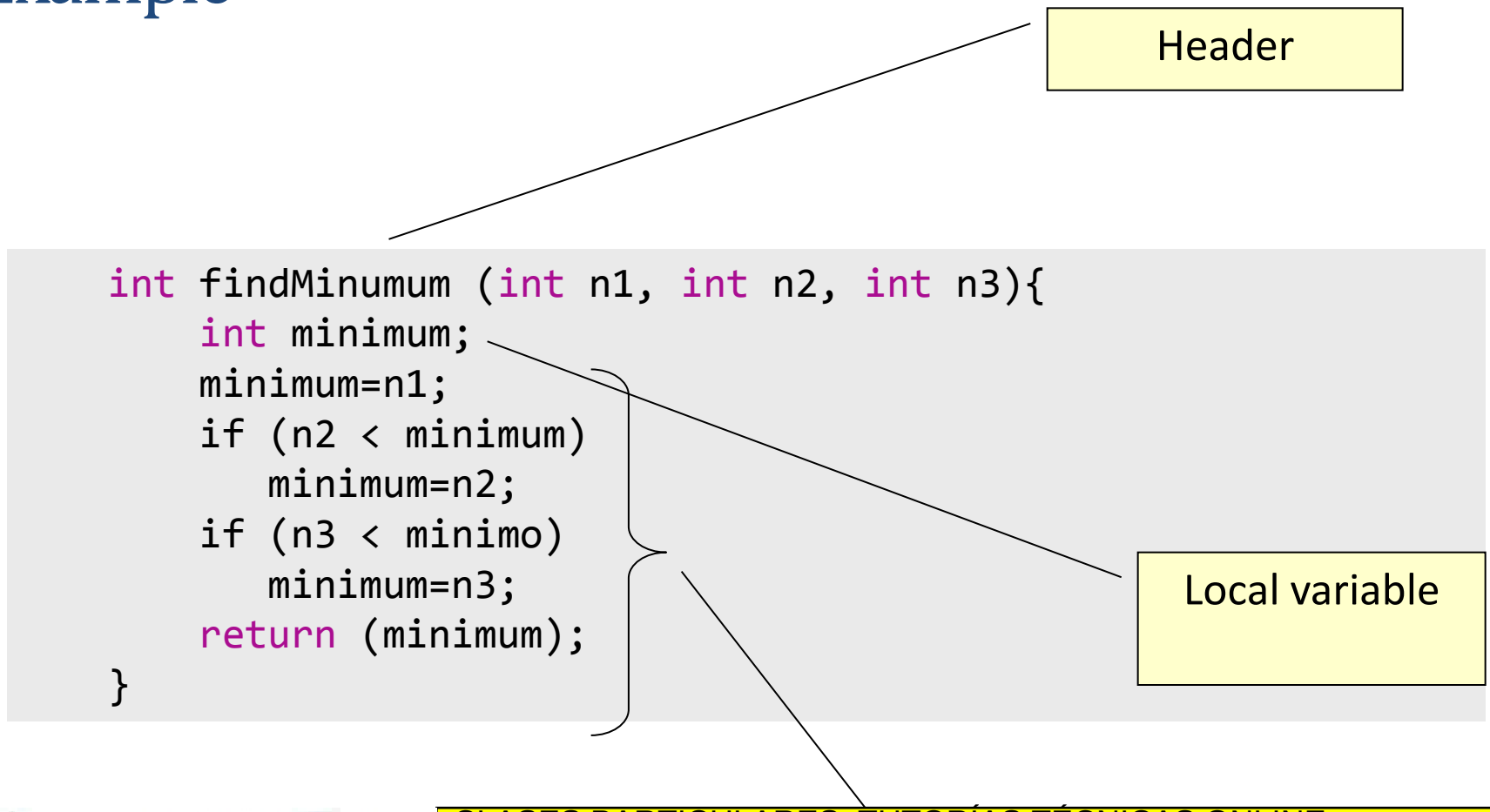
- The instruction **return** returns control to the function that made the call
  - If there were any other instructions after the return, they would never be run

```
Style rules for our course: include only one return instruction, that will be the last instruction in your function, following the principles of structured programming
```

Local variables

- A function may have it's own variables
  - We call them **local variables**
  - They are declared at the beginning of the function's body
  - They are visible only within the block where they are declared, invisible to the rest of the program
  - They raise into existence
  - Local variables come to existence when the function is called
  - They cease to exist when the function ends (return)

Example



Function declaration: prototype

- Before using a function, we need to declare it
  - Same as we do with variables
  - To do this we use the **prototype**
  - Prototype reports the existence of a function, and that the details of how the function works will be found elsewhere
  - The prototype must be found in the code before the function is used
    - Usually, at the beginning of the program (after #include - #define) and before the main function
  - The prototype is identical to the function header but ending with

Function declaration : examples

```
int findMinimum (int n1, int n2, int n3);

int calcPower (int base, int exponent);

float add(float n1, float n2);

void displayData(int a, int b);

int getData(void);
```

5.3 FUNCTION CALLING

Calling a function

- A function is called (invoked) when it's name is used in an expression or instruction
- examples

When the function is void (no return value)

```
min=findMinimum(a, b, c);
min=findMinimum(3, 17, -2);
printf("%d", findMinimum, b, c));

printList(data);
```

- When the function is called, the instructions within the function are executed

Calling a function

- A function call can be done from the main function or from within another function
- Function name is followed by a list of parameters
  - Divided by comas, within parenthesis
  - If the function doesn't take parameters, only parenthesis with nothing in ()

- When the function is called, the program evaluates the parameters, and passes a copy of the values to the function, handling the execution control to the function

```c
#include <stdio.h>
int findMinumum (int n1, int n2, int n3);

int main (void){
    int num1, num2, num3;
    int min;
    printf ("Enter three integer values: \n");
    scanf ("%d", &num1);
    scanf ("%d", &num2);
    scanf ("%d", &num3);
    min= findMinumum (num1, num2, num3);
    printf ("The smallest of the three is %d \n", min);
    system ("PAUSE");
    return 0;
}


     int findMinumum (int n1, int n2, int n3){
         int minimum;
         minimum=n1;
         if (n2 < minimum)
            minimum=n2;
```

Calling a function

- **Actual parameters**
  - Parameters in the function **call**
  - Can be variables, constants, literals or expressions
- **Formales parameters**
  - Parameters in the function definition
  - Can only be variables

- Correspondence among parameters is based on their order
  - `Function call min=findMinimum(3, 17, -2);`
  - `Function definition int findMinumum (int n1, int n2, int n3){`

3. Function calling

```
int findMinumum (int n1, int n2, int n3);

int main (void){
    int num1, num2, num3;
    int min;

    printf ("Enter 3 values: \n");
    scanf ("%d", &num1);
    scanf ("%d", &num2);
    scanf ("%d", &num3);
    min1=findMinumum (num1, num2, num3);
      3              3    50    25

printf ("The smallest of the three is %d \n", min);
    system ("PAUSE");
    return 0;
}
                          3    50    25

     int findMinumum (int n1, int n2, int n3){
         int minimum;
         minimum=n1;
```

Actual parameters

Calling a function

- Actual and formal parameters have to be
  - **Same number**
    - Same number of parameters in the function declaration (formal) and function call (actual)
  - **Same data type**
    - Each parameter in the declaration (formal) must be of the same datatype as the corresponding parameter in the function call (actual)

3. Function calling

```
#include <stdio.h>

int findMinumum (int n1, int n2, int n3);

int main (void){
    /*Minumum of 9 numbers*/
    /*Using find minimum of three numbers*/
    int num1, num2, num3;
    int min1, min2, min3;
    int min;

    printf ("Enter 9 values: \n");
    scanf ("%d", &num1);
    scanf ("%d", &num2);
    scanf ("%d", &num3);
    min1=findMinumum (num1, num2, num3);

    scanf ("%d", &num1);
    scanf ("%d", &num2);
    scanf ("%d", &num3);
    min2=findMinumum  (num1, num2, num3);

    scanf ("%d", &num1);
    scanf ("%d", &num2);
    scanf ("%d", &num3);
    min3=findMinumum (num1, num2, num3);

    return 0;
```

Prototype

Formal parameters

Return value

Function call

Actual parameters

5.4. PASSING PARAMETERS TO FUNCTIONS: BY VALUE AND BY REFERENCE

Passing parameters

- Summary from previous slides
  - When passing parameters to a function a correspondence between the parameter in the call (actual) and in the declaration (formal) is set
    - Input data are passed to the function
  - This correspondence is based on the position
  - Number of parameters and datatypes must match

Pass by value

- The function gets a **copy of the values** in the calling function
  - This copy is stored in the formal parameter (in the function parameter)
- The function operates on the formal parameter
  - Any changes made are only made to the copy, not to the original variable
  - If the value changes, this has no effect outside the function

- When to use pass by value
  - When the function doesn't modify the parameters (input data)
  - When we don't want the changes made by the function to affect

Pass by reference

- The function gets a **reference to the memory address** where the data to use is stored
  - Not a new variable with a copy of the value as in pass by value
  - This is done by using the address of the memory cell allocated to the variable (a pointer to the variable)

- After the function call returns, any change to the parameter is seen from the main program
  - This can be seen as output data: the function can return values to the main function using parameters passed by reference

Pass by reference: syntax

- In the main function (call):
  - The parameter is preceded by the **address-of operator (&)**, indicating that what is passed to the function is the memory address where the variable is stored:

    `&var1`

    As seen with scanf !
- In the prototype, declaration and function body:
  - The formal parameter is preceded by the **indirection operator (*)**, indicating we access to the contents pointed at by the variable

    `(int*param1)`

- As we work with pointers we access the real value of the

Pass by reference, example 1

```
void increase (int *a);

int main (void){
   int var1=1;
 increase(&var1);
   return 0;
}

void increase (int *a){
   *a=*a + 1;
   return;
}
```

Actual parameter is a reference to the memory address where the data is stored
(& var1)

Formal parameter: the function gets the memory address where the variable is stored. That's why it the formal parameter is declared here as a **pointer** to the variable

To access the real parameter we use the **indirection operator** (content of) on the

Pass by reference, example 2

```
#include <stdio.h>
void swap*x, int *y);

int main(void)
{
    int num1=3;
    int num2=50;
    printf("num 1 is %i and num 2 is %i ", num1, num 2);

    swap(&num1, &num2);
    printf("num1=%i num2=%i", num1, num 2);
    return 0;
}

void swap(int *x, int *y)
{
    int aux;
    aux=*x; //Step 1. aux takes the value "pointed at" by x
    *x=*y;  //Step 2. *x takes the value of *y
```

Actual parameters: memory address where the data to modify are stored (adress-of operator : "**&**")

Formal parameters: Declared as pointers, they receive the memory address where the data are stored – to access the data itself we need to use "content of" *

Keyword *const* for function parameters

```
#include <stdio.h>

int suma(int a, int b);

int main(void)
{
   int n1, n2, resu;

   printf ("Give me two values\n");
   scanf("%i",&n1);

   scanf("%i",&n2);

 //we add them using a function, just as an example
   resu=suma(n1,n2);


   printf("sum of %i and %i is %i", n1, n2, resu);
   return 0;
}

int suma (const int a, const int b)
{
```

Tells the compiler that these

thrown

5.5 SCOPE OF VARIABLES AND VISIBILITY

Scope of a variable declaration

- Scope of a variable is the section of code in which the variable is valid, i.e. where it can be accessed and used
  - **Locals** variables : within a function
  - **Globals** Variables : from all the program

- Local variables :
  - Declared inside a function – at the beginning of a code block.
  - Only visible within that block of code (the function)
  - Formal parameters have the same scope as a local variable (within the function in which they are declared

- Can be accessed from everywhere in the program

Ámbito de declaración de variables: Visibilidad

```
(Function) Parameter y: 1
(Function) local Variable x: 4
(Main) Variable a in the main function: 1
(Main) Local variable x: 3
```

```c
#include <stdio.h>

void f(int y);

int main(void) {
        int a = 1, b = 2, x = 3;

        f(a);
        printf("(Main) Variable a in the main function: %i\n", a);
        printf("(Main) Variable x in the main function : %i\n", x);

        return 0;
}

void f(int y) {
        int x = 4;

        return;
}
```

This **x** es is a local variable to **main**

This **x** is a local variable to function f**f, and is different from x** in**main**

Scope of a variable declaration

- **Good practice:**
  - AVOID USING GLOBAL VARIABLES
  - ANYTIME A FUNCTION HAS TO USE A VARIABLE FROM THE MAIN FUNCTION OR FROM OTHER FUNCTION IT HAS TO TAKE IT AS A PARAMETER
    - Even if the variable is visible within the function
  - Why: good quality code
    - Better readability, easier to understand, upgrade and debug
      - Other programmers in the team can follow the code
    - Less mistakes are made and errors are easier to find

ANNEX
STANDARD LIBRARIES IN C

Libraries of functions

- The main function and the rest of the functions can be in the same file or different files
- Grouping functions in files according to their type facilitates reuse: Libraries of functions
- Libraries comprise two files
  - Header file .h – containing the function declaration (prototype)
    - They may also contain constants and structures (Unit 7)
  - Source file .c – containing the function definition (código)

```
#include "MyFunctions.h"
```

Header file containing

Standard C libraries

- C language provides **several standard libraries with functions** implementing common tasks
- Each library groups functions of the same type (input-output, mathematical, with strings)
  - We've already used some
    - stdio.h, math.h, string.h

- To use a function we must include it's prototype in the code
  - Just as the functions we create
  - To do so we include the header file (.h) where all the functions in

Some useful standard libraries

<complex.h>	Complex numbers arithmetic's
<ctype.h>	character management
<errno.h>	Error control
<float.h>	additional functionality for dealing with float number
<math.h>	Mathematical functions
<stdio.h>	standard input output - io
<stdlib.h>	absolute value, random number generation, search and sort, string conversion, memory management and itnerface with the operation g system
<string.h>	string management

Commonly used functions

Function	Returns	Action	Library
abs(i)	int	Absolute value of **i**	stdlib.h
fmod(d1, d2)	double	Module of the division **d1**/**d2** (with **d1** sign)	math.h
sqrt(d)	double	Square root of **d**	math.h
atoi(s)	long	String **s** is converted into an integer value	stdlib.h
atof(s)	double	String **s** is converted into a real value	stdlib.h
floor(d)	double	Largest integer not greater than **d**, as a double	math.h
ceil(d)	double	Smallest integer not less than **d**, as a double	math.h
exp(d)	double	Exponential function	math.h
log(d)	double	Natural logarithm (**d** > 0)	math.h
rand(void)	int	Pseudo-random integer in the range 0 to RAND_MAX	stdlib.h
sin(d)	double	Sine of **d** (in radians)	math.h
cos(d)	double	Cosine of **d** (in radians)	math.h
tan(d)	double	Tangent of **d** (in radians)	math.h
asin(x)	double	Sin^{-1} of **x**	math.h
strcmp(s1, s2)	int	Compares **s1** and **s2**; if equal, it returns 0	string.h

strcpy and strcat

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAM_CADENA   80

int main (void)
{
  //Variable declaration
  char nombre[TAM_CADENA];
  char apellidos[TAM_CADENA];
  char nombreCompleto[TAM_CADENA*2];

  printf ("Enter your name: \n");
  scanf("%s", nombre);
```

example (cont.)

```
/* Almacenemos en nombreCompleto el nombre y los apellidos*/
/* 1. inicia nombreCompleto a la cadena vacía */
strcpy (nombreCompleto, "");

/*2. concatena el nombre*/
strcat(nombreCompleto, nombre);

/*3. concatena un espacio en blanco*/
strcat(nombreCompleto, " ");

/*4. concatena los apellidos*/
strcat(nombreCompleto, apellidos);

/*5. Se imprime el nombre completo*/
printf("Your full name is : %s\n", nombreCompleto);
```

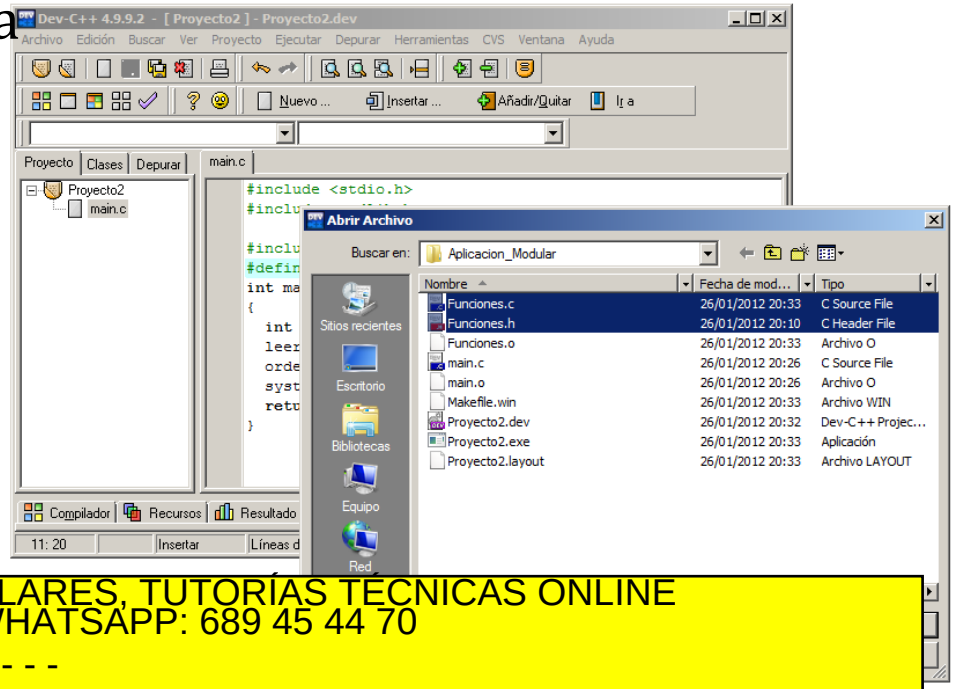
Comparing strings (strcmp)

```
int main(void) {
int result;
// Create two arrays to hold our data
char example1[50];
char example2[50];
// Copy two strings into our data arrays
strcpy(example1, "C programming is useful");
strcpy(example2, "C programming is fun");
// Compare the two strings provided
result = strcmp(example1, example2);
// If the two strings are the same say so */
 if (result == 0)
      printf("Strings are the same\n");
```

ANNEX
LIBRARY FUNCTIONS WITH DEVCPP

How to create your own library in DevC++

- Step 1. Create your functions
  - Prototypes in a file with filename extension.h
  - Definitions (funcition code) in a file with filename extension .c
- Step 2. Add to project
  - Add both files .c and .h to your project

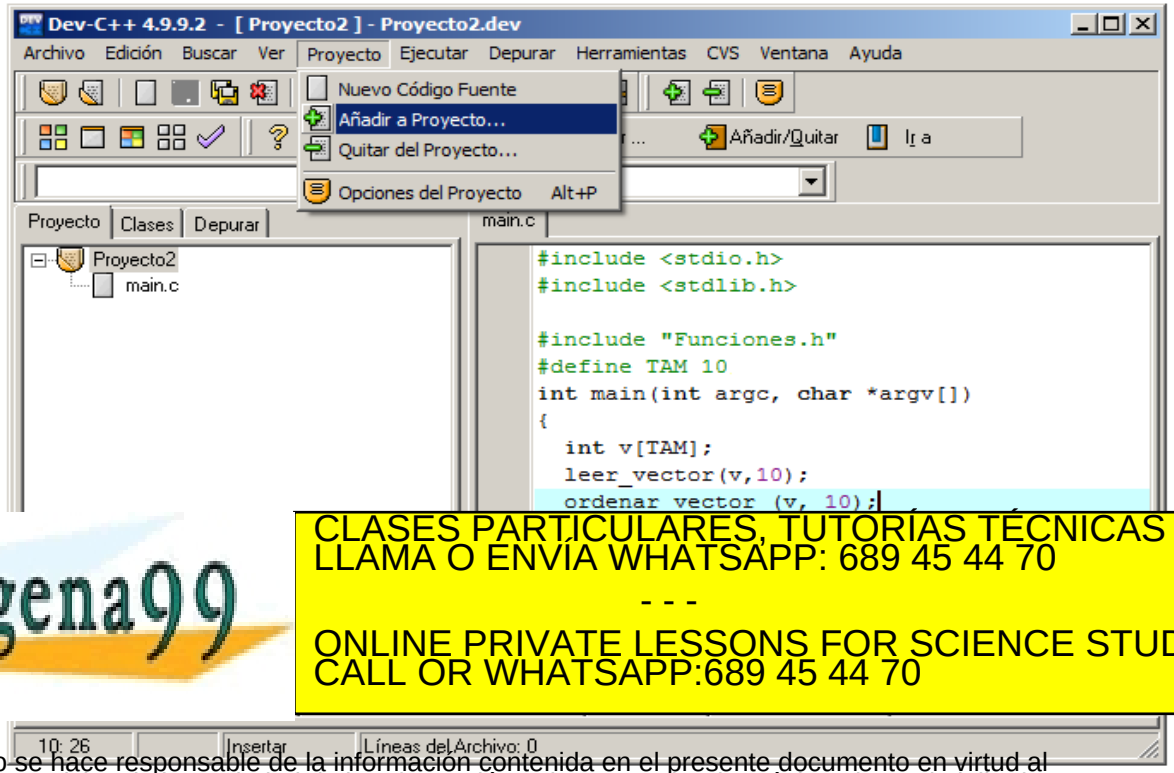


How to create your own library in DevC++

Step 3. Write the main function

- Including the header file

```
#include "funciones.h“
```



How to create your own library in DevC++

Step 4. Compile:

Different compiler options, make sure you complie all files needed

- ✓ **Execute -> Compile**: Compiles only the files modified since the last time the project was compiled
- ✓ **Execute -> Compile current file**: Compiles only the current file
- ✓ **Execute -> Rebuild all**: Compiles all the files in the project

UNIT 5. FUNCTIONS

Programming

Year 2017-2018

Grade in Industrial Technology Engineering