jgromero@inf.uc3m.es

Lesson 6
Functions

Programming

Grade in Industrial Technology Engineering

Outline

1. **Modular programming**
2. **Function declaration and definition**
3. **Function calling**
4. **Parameters – Call by value and by reference**
5. **Parameters – Arrays and structures**
6. **Scope of variables in functions**
7. **Library functions**

jgromero@inf.uc3m.es

1. **Modular programming**
2. Function declaration and definition
3. Function calling
4. Parameters – Call by value and by reference
5. Parameters – Arrays and structures
6. Scope of variables in functions
7. Library functions

Programming paradigm

Set of programming techniques to create *good* programs

Recommendations, blueprints, patterns, etc. supported by language constructs

What is a **good program**?

Correct

The program calculates correct results

Easy to debug

The program facilitates error location and solving

Easy to extend

The program facilitates adding new functionalities

Readable

The program can be easily understood by any other programmer

Well-documented

The program includes comments and additional documentation to support other programmers

Paradigms

- Conventional programming
- Structured programming
- **Modular programming**
- Object-oriented programming
- Component-based programming
- …

Conventional programming

No programming methodology is used

> **Result**: Large programs which are difficult to read and maintain

Structured programming

The program has a unique starting point and a unique ending point

Use of a restricted set of control instructions: sequential, conditional and loops (jumps/goto instructions are not allowed) [Böhm-Jacopini theorem]

> **Result**: Programs are easier to read, but still monolithic

Modular programming

It is based on decomposing the solution of a large problem into smaller solutions (*modules* or *subprograms*) which are easier to analyze, develop, and debug

The solution to the main problem is obtained by combining solutions calculated by the modules

> **Result**: Programs which are more readable and easier to deal with

Modular programming

A program includes:

The main program

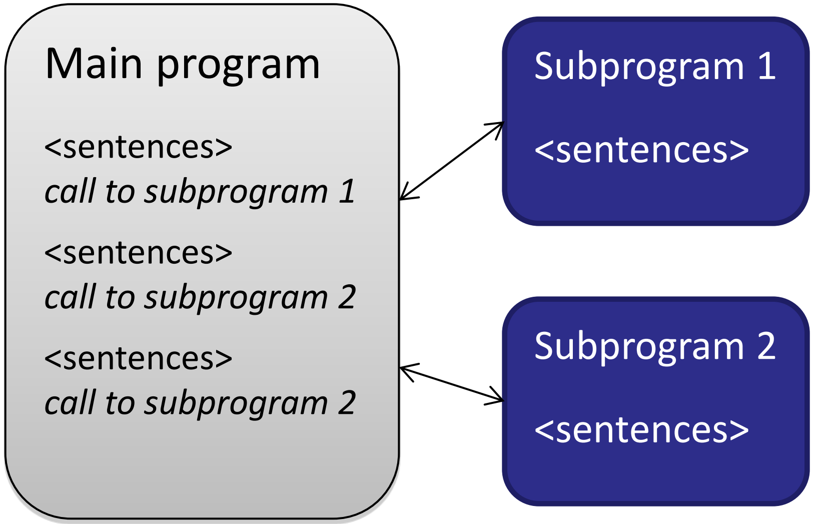
Basic operations and calls to the subprograms

(In C, it is the `main` function)

Subprograms

Independent units to solve a particular problem

(In C, they are called *functions*)



1. Modular programming

Advantages

> **Programs are easier to read**

Modules are shorter and simpler, since they calculate a partial solution of the problem

> **Programs are easier to test and debug**

Modules can be developed and tested individually

Different programmers can work in different parts of the program

> **Programs are easier to maintain and extend**

A module of the program can be modified without affecting other modules of the program

> **Programs are reusable**

Functions and modules can be reused in programs that require similar functionalities

C promotes modular programming

A **function** is a self-contained unit of program code designed to accomplish a particular task

A C program is composed of several functions

Functions are called (or invoked) from the main function or from any other function

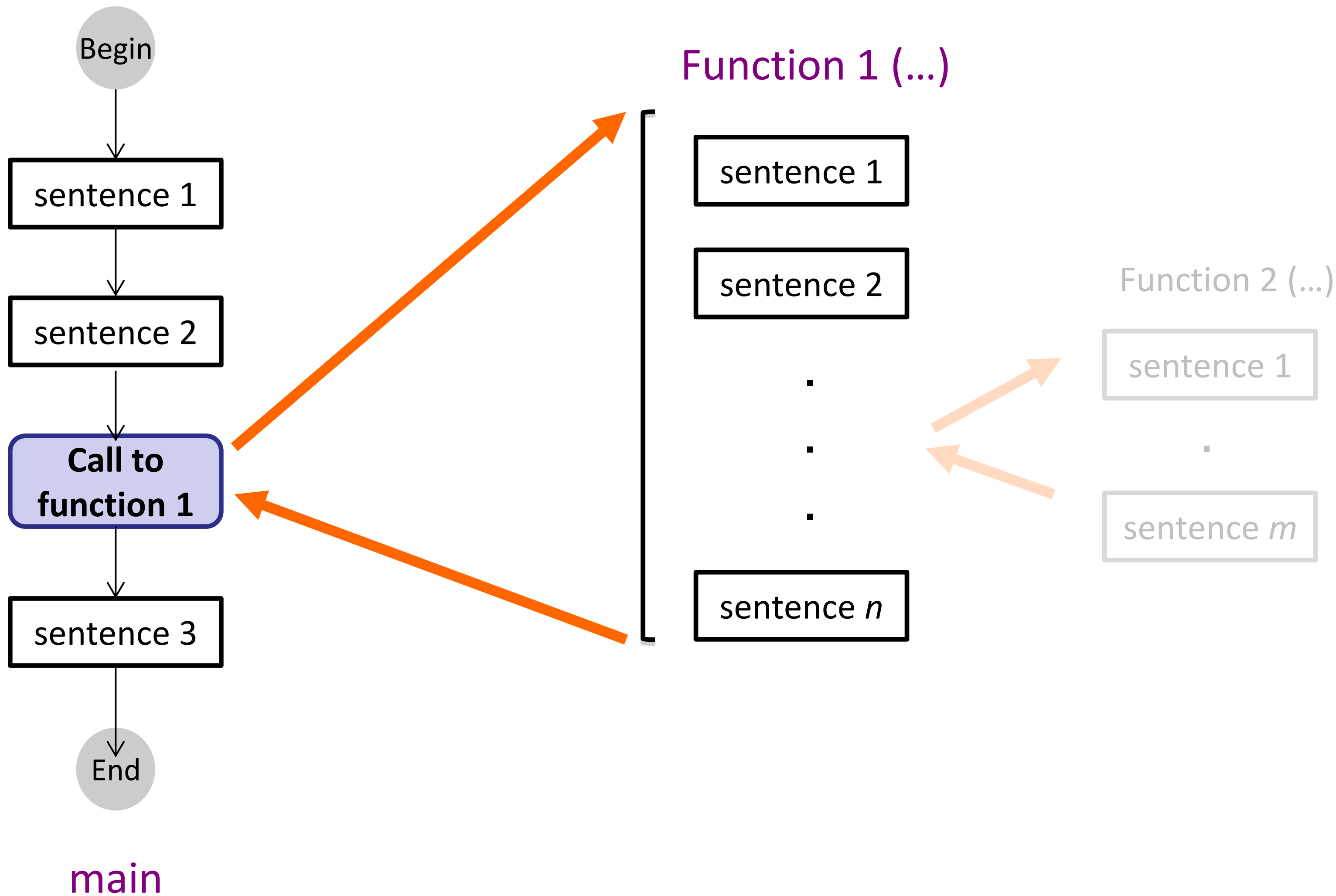
The task function provide a resulting value to the calling function –which may be the `main` or any other function

A collection of functions is a library

Designing a proper set of functions is fundamental to create a good program

1. Modular programming

Modular programming in C



Outline

1. Modular programming
2. **Function declaration and definition**
3. Function calling
4. Arguments – Call by value and by reference
5. Arguments – Arrays and structures
6. Scope of variables in functions
7. Library functions

The **declaration** of the function specifies the **prototype**

name of the function,

type of the returned value,

type of the *parameters*

a parameter is a value that is used by the function

the value of the parameter is specified when the function is called

Syntax

```
<type> <name> (<list of parameters with types>);
```

2. Function declaration and definition
Function declaration

<type>

Type of the value returned by the function
It cannot be an array
However, it can be a pointer –remember the correspondence between arrays and pointers
If the function does not return any value, the returning type is `void`

<name>

Name of the function

<list of parameters>

Parameters that this function receives (**formal parameters**)
The type of each one of the parameters must be specified
The name is optional, but convenient
If the function does not receive any parameter, it must be indicated with `void`

Examples

```
float power(float base, int exponent);
float add(float x, float y);
void print(int l[]);
int read(void);
```

Notice!

The prototype of the function indicates that a function with the specified properties (returning type, name, arguments) will be defined below

All functions (except the `main`) must be declared (i.e., the prototype must be specified) before calling them

Therefore, it is convenient to define the functions just before the `main`

The prototype of a function ends with `;`

A **function definition** is **the code to perform the intended task**

Function definition has two parts:

- **Function header**
  - Similar to the prototype, but without an ending ;

```
float add(float x, float y)
```

- **Function body**
  - Sentences to perform the task that are executed when the function is called
  - Local variables may be defined
    - The scope of these variables is the function (they cannot be used from any other function)
    - They are allocated in each call to the function

```
{ float result;
  result = x + y;
  return result; }
```

Syntax

```
<type> <name> (<list of parameters>) {
    <variable declarations>
    <sentences>
    return <expression with proper type>
}
```

Header

Local variables

End function and return a value

The definition can be placed at any point of the file, with the following restrictions:

- A definition must be after the declaration of the function
- A definition cannot be inside another function definition (particularly, it cannot be inside in the `main` function)

2. Function declaration and definition

Example

```c
#include <stdio.h>

float add(float x, float y);

int main(void) {

    /* Implementation of main method */

    return 0;
}

/* Definition of add function */
float add(float x, float y) {
    float result;
    result = x + y;
    return result;
}
```

Declaration

Definition

Function header

Local variable

Function body

A function must return a value as a result (the *returning value*)

This value is used in the function that calls to this function

The type of the returning value is specified in the prototype and in the definition header

- `void` is used for functions that do not return a value

The `return` instruction (inside the function):

- Ends the function
- Provides the returning value as the result of the function

Recommendation

Use only one `return` in a function and place it at the end of the function

Outline

1. Modular programming
2. Function declaration and definition
3. **Function calling**
4. Arguments – Call by value and by reference
5. Arguments – Arrays and structures
6. Scope of variables in functions
7. Library functions

Calling or invoking a function means "*execute the instructions of the function and obtain the returning value*"

To call a function, **use**:

- Name of the function
- Opening parenthesis
- List of values (or expressions) separated by commas (**actual parameters or arguments**)
- Closing parenthesis

When the program execution reaches a call to a function, the execution steps into the function

The expressions in the call are evaluated and the values are assigned to the parameters

- The number and the type of the arguments in the call must match the number and the type of the parameters specified in the prototype

```
float c;
c = add(a, b*0.75);        // a and b are float variables
printf("%f", add(a, b));   // the result of add is not stored, just printed
```

3. Function calling

Example

Declaration

Definition

Formal parameters

Actual parameters

Call to the function

```c
#include <stdio.h>

float add(float x, float y);

int main(void) {

    float n1, n2, r;

    /* Read values */
    printf("Enter two numbers: ");
    scanf("%f %f", &n1, &n2);

    /* Use function */
    r = add(n1, n2);

    /* Print result */
    printf("%f + %f --> %f", n1, n2, r);

    return 0;
}

/* Definition of add function */
float add(float x, float y) {
    float result;
    result = x + y;
    return result;
}
```

3. Function calling

Example

```
#include <stdio.h>

float add(float x, float y);
float power(float base, int exponent);

int main(void) {

    float n1, n2, r;

    /* Read values */
    printf("Enter two numbers: ");
    scanf("%f %f", &n1, &n2);

    /* Use functions */
    r = add(n1, n2);
    printf("%f + %f --> %f\n", n1, n2, r);

    r = power(add(n1, n2), 2);
    printf("(%f + %f)^2 --> %f", n1, n2, r);

    return 0;
}
```

```
/* Definition of add function */
float add(float x, float y) {
    float result;
    result = x + y;
    return result;
}

/* Definition of power function */
float power(float base, int exponent) {
    float result = 1;
    int i;

    for(i=1; i<=exponent; i++)
        result *= base;

    return result;
}
```

Outline

1. Modular programming
2. Function declaration and definition
3. Function calling
4. **Parameters – Call by value and by reference**
5. Parameters – Arrays and structures
6. Scope of variables in functions
7. Library functions

Parameters are used to define general functions that can operate with different input data

- E.g.: `power` function should be able to calculate the power x^n for any value of x and n

How these arguments are passed to the functions?

- **Formal parameters** are defined in the function prototype
  - Can be considered as variables local to the function that are created at the beginning of the function and destroyed at the end
- **Actual parameters or arguments** are provided in the call to the function
  - The first actual parameter corresponds to the first formal parameter, and so forth.

4. Parameters – Call by value and by reference

Call by value

The function **stores a copy** of the values of the actual parameters into the formal parameters

If the value of a parameter is changed inside a function, this change has no effect out of the function

To use a value calculated inside the function, use **`return`** instruction to return the value

(This calculation may involve assigning values to the function parameters inside the function)

4. Parameters – Call by value and by reference

Call by value

```
#include <stdio.h>

int add(int a, int b);

int main(void) {

        int n1, n2, res;

        printf("Enter two values:\n");

        scanf("%i",&n1);
        scanf(”%i",&n2);

        res = add(n1,n2);

        printf(“%i + %i --> %i", n1, n2, res);

        return 0;
}

int add(int a, int b) {
        int r;
        r = a+b;
        return r;
}
```

```
Enter two values:
10
15
10 + 15 --> 25
```

Universidad Carlos III de Madrid
www.uc3m.es

4. Parameters – Call by value and by reference

Call by value

Memory Address	Step 0 **int** n1, n2	Step 1 (scanf)	Step 2	Step 3 r= a+b	Step 4
0012FF88	**n1**	**n1**	**n1**	**n1**	**n1**
	?	10	10	10	10
0012FF84	**n2**	**n2**	**n2**	**n2**	**n1**
	?	15	15	15	15
0012FF80	**res**	**res**	**res**	**res**	**res**
	?	?	?	?	25
0012FF7C			**a**	**a**	
			10	10	
0012FF78			**b**	**b**	
			15	15	
0012FF70			**r**	**r**	
			?	25	

main() variables: 0012FF88 – 0012FF80

add() parameters: 0012FF7C – 0012FF70

call to add() → Step 2

main() variables are not accessible (Steps 2–3)

return to main() → Step 4

Universidad Carlos III de Madrid
www.uc3m.es

4. Parameters – Call by value and by reference

Call by value

```
#include <stdio.h>

float add(float a, float b);

int main(void) {

        float n1, n2, res;

        printf("Enter two values:\n");

        scanf("%f",&n1);
        scanf(”%f",&n2);


        res = add(n1,n2);

        printf(“%f + %f --> %f", n1, n2, res);

        return 0;
}


float add(float a, float b) {
        a = a+b;
        return a;
}
```

```
Enter two values:
10
15
10.0 + 15.0 --> 25.0
```

Assigning a value to a function parameter

Universidad Carlos III de Madrid
www.uc3m.es

4. Parameters – Call by value and by reference

Call by value

```
Before calling demo function: n --> 10
Inside demo function: value --> 10
Inside demo function: value --> 999
After calling demo function: n --> 10
```

```c
/* Example: Passing parameters by value */

#include <stdio.h>
void demoFunction1(int value);

int main(void) {
        int n=10;

        printf("Before calling demo function: n --> %i\n", n);
        demoFunction1(n);
        printf("After calling demo function: n --> %i\n",n);

        return 0;
}

void demoFunction1(int value) {
        printf("Inside demo function: value --> %i\n", value);
        value= 999;
        printf("Inside demo function: value --> %i\n", value);
}
```

4. Parameters – Call by value and by reference

Call by reference

Call by reference is used if we want to change the value of a parameter **and keep this change out of the function**

The underlying idea is to pass *the address* of the variable, instead of the value of the variable, and change indirectly this value

To pass a parameter by reference, it is required:
[**NOT** for arrays –see later]

& in the actual parameter (call to the function

* in the forma parameter (definition of the function)

Call by reference must be used if more than one value must be changed by a function

4. Parameters – Call by value and by reference

Call by reference

Remember:

address-of operator (`&`)

obtains the memory address of the variable

`x = &a;` *x* stores the address of *a* contents

(*x* must be a pointer to type of *a*)

indirection operator (`*`)

obtains the value stored in the address stored in a pointer

`*x = 10;` stores value 10 at the address pointed by *x*

(*x* must be a pointer to an integer)

4. Parameters – Call by value and by reference

Call by reference

```
#include <stdio.h>

void increment(int *a);

int main(void) {

        int v=1;

        printf("v-->%i", v);
        increment(&v);
        printf("v-->%i", v);

        return 0;
}

void increment(int* a) {

        *a = *a + 1;
        return;

}
```

Actual parameters: Reference to the memory address of the variables that will be modified (**&** precedes the variable name)

Formal parameters: Store the address of the formal parameters that will be modified (***** precedes the variable name)

Indirection: Indirection operator is used to access to the variable

```
v-->1
v-->2
```

4. Parameters – Call by value and by reference

Call by reference

```
#include <stdio.h>

void swap(int *x, int *y);

int main(void ) {

        int i=3;
        int j=50;

        printf("i-->%i, j-->%i\n", i, j);

        swap(&i, &j);
        printf("i-->%i, j-->%i\n", i, j);

        return 0;
}

void swap(int* x, int* y) {
        int aux;
        aux=*x;   // Step 1. The value at the address pointed by x (*x) is assigned to aux
        *x=*y;    // Step 2. The value at the address pointed by y is
                  // assigned to the address pointed by x
        *y=aux;   // Step 3. aux is assigned to the address pointed by y
}
```

Actual parameters: Reference to the memory address of the variables that will be modified (**&** precedes the variable name)

Formal parameters: Store the address of the formal parameters that will be modified (* precedes the variable name)

Universidad Carlos III de Madrid
www.uc3m.es

4. Parameters – Call by value and by reference

Call by reference

	Memory address		Step 1 (aux=*x)	Step 2 (*x=*y)	Step 3 (*y=aux)	
main() variables	0012FF88	i				i
		3	3	**50**	50	**50**
	0012FF84	j				j
		50	50	50	**3**	**3**
swap() parameters	0012FF7C		x* x	x	x	
			0012FF88	0012FF88	0012FF88	
	0012FF78		y	y* y	y	
			0012FF84	0012FF84	0012FF84	
	0012FF70		aux	aux	aux	
			3	3	3	

Call to swap(int, int*)*

main() variables are not **directly** accessible

Return to main()

4. Parameters – Call by value and by reference

Highlights

Differences between passing arguments modes:

By value

- The value of the actual parameter is copied into the formal parameter
- Values assigned to the formal parameters inside the function does not change the value of the actual parameter in the function call
- Only one value can be returned

By reference

- Formal parameters are declared as pointers; actual parameters are variable addresses (& operator) –except for arrays
- Inside the function, the values of the actual parameters are changed by using indirect access with pointers (* operator)
- Only one value can be returned, but several parameter values can be modified

4. Parameters – Call by value and by reference

`const` parameters

To avoid bad use of functions, the **`const`** specifier may be used to define a formal parameter as a read-only one

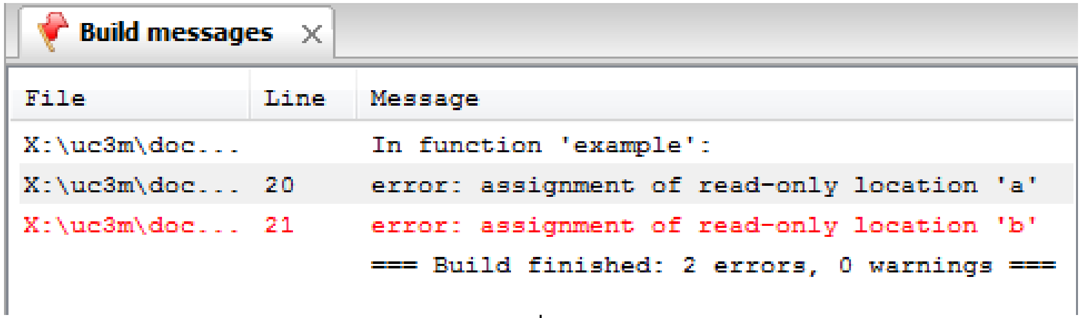
```
// Function declaration
float add(const float a, const float b);
…


// Function definition
float add(const float a, const float b) {
    …
}
```

`const` forces the compiler to give an error if a parameter defined as `const` is changed inside the function

4. Parameters – Call by value and by reference

`const` parameters

```c
#include <stdio.h>

void example(const int a, const int b);

int main(void) {
    int n1, n2;
    printf("Enter two values:\n");
    scanf("%i",&n1);
    scanf("%i",&n2);

    printf("%i + %i --> %i", n1, n2);
    example(n1,n2);
    printf("%i + %i --> %i", n1, n2);

    return 0;
}

void example(const int a, const int b) {
        a = 12;
        b = 13;
}
```



Build messages

File	Line	Message
X:\uc3m\doc...		In function 'example':
X:\uc3m\doc...	20	error: assignment of read-only location 'a'
X:\uc3m\doc...	21	error: assignment of read-only location 'b'
		=== Build finished: 2 errors, 0 warnings ===

4. Parameters – Call by value and by reference

`const` parameters

```
#include <stdio.h>

float add(const float a, const float b);

int main(void) {
          float n1, n2, res;

          printf("Enter two values:\n");
          scanf("%f",&n1);
          scanf("%f",&n2);

          res = add(n1,n2);
          printf("%f + %f --> %f", n1, n2, res);

          return 0;
}


float add(const float a, const float b) {
          float r;
          r = a+b;
          return r;
}
```

Outline

1. Modular programming
2. Function declaration and definition
3. Function calling
4. Parameters – Call by value and by reference
5. **Parameters – Arrays and structures**
6. Scope of variables in functions
7. Library functions

An array can be the parameter of a function

Function declaration

In the prototype:

- Type of the array
- Name of the formal parameter (the name that will be used within the function)
- Brackets [ ]   **SIZE** is optional (not recommended to use it)

The number of (valid) elements of the array must be also passed as argument

array without size

number of elements of the array

Function prototype

```
#define MAX 100

int addData(int a[], int length);
```

An array can be the argument of a function

Function calling

In the call to the function:

- Array name (**without brackets!**)
- Number of elements

```
int main(void) {

    int d[MAX];
    ...
    add = addData(d, MAX);
    ...
}
```

Call to the function

Passing an array as parameter means *passing the address of the first element of the array*

- The array itself is passed by value
- Array elements are (kind of) **passed by reference**
  - The **values of the elements** of the array can be changed inside the function
  - It is not necessary to use the `&` operator before the name of the array in the call to the function to change the values of the elements
- To indicate that the function does not modify the array elements, it is recommended to use the `const` modifier

```
int addData(const int a[], int length);
void readArray(int a[], int length);
```

```c
#include <stdio.h>
#define SIZE 5

int addData(const int a[], int length);
void readArray(int a[], int length);

int main(void) {
    int v[SIZE];

    printf("Enter vector values\n");
    readArray(v, SIZE);
    printf("Add values --> %i \n", addData(v, SIZE));

    return 0;
}

int addData(int a[], int length) {
    int i, sum = 0;
    for (i=0; i<length; i++)
         sum += a[i];
    return sum;
}

void readArray(int a[], int length) {
    int i;
    for (i=0; i<length; i++)
         scanf("%i", &a[i]);
}
```

Actually, passing the number of elements of the array as parameter is very convenient, but no compulsory

```
#include <stdio.h>
#define SIZE 5

void readArray(int a[]);

int main(void) {
      int v[SIZE];
      printf("Enter vector values\n");
      readArray(v);
      return 0;
}

void readArray(int a[]) {
      int i;
      for (i=0; i<SIZE; i++)
           scanf("%i", &a[i]);
}
```

readArray can be **ONLY** used with **`int`** arrays with length **SIZE**

jgromero@inf.uc3m.es

5. Parameters – Arrays and structures

Arrays as parameters

guments

```
#include <stdio.h>
#define SIZE 5

void readArray(int a[], int length);

int main(void) {
    int v[SIZE];
    printf("Enter vector values\n");
    readArray(v, SIZE);
    return 0;
}


void readArray(int a[], int length) {
    int i;
    for (i=0; i<length; i++)
        scanf("%i", &a[i]);
}
```

readArray can be used with **`int`** arrays of any **length**

5. Parameters – Arrays and structures

Arrays as parameters

Write a C program that creates two 1-dimension arrays of integer values, copies the values of these arrays in a third array, and print the values of all of them on the screen by using functions

```c
#include <stdio.h>
#define SIZE_1 5
#define SIZE_2 3

void readArray(int a[], int n);
void printArray(const int a[], int n);
void copyArrays(const int a[], const int b[], int c[], int n1, int n2);

int main(void) {
     int v1[SIZE_1], v2[SIZE_2], v3[SIZE_1 + SIZE_2];

     printf("Enter array 1 values: \n");
     readArray(v1, SIZE_1);
     printf("Enter array 2 values: \n");
     readArray(v2, SIZE_2);

     copyArrays(v1, v2, v3, SIZE_1, SIZE_2);

     printf("Array values: \n");
     printArray(v1, SIZE_1);
     printArray(v2, SIZE_2);
     printArray(v3, SIZE_1 + SIZE_2);

     return 0;
}
```

```
void readArray(int a[], int n) {
   int i;

   for (i=0; i<n; i++)
          scanf("%i", &a[i]);
}

void printArray(const int a[], int n) {
   int i;

   printf("[ ");
   for (i=0; i<n; i++)
          printf ("%i ", a[i]);
   printf("] \n");
}

void copyArrays(const int a[], const int b[], int c[], int n1, int n2) {
   int i;

   for (i=0; i< n1+n2; i++) {
          if (i<n1)
                c[i]=a[i];
          else
                c[i]=b[i-n1];
   }
}
```

Multiple-dimension arrays

Passing multiple-dimension arrays as parameters is not a direct extension of the case for one-dimension arrays

To declare (and define) a function with two-dimension arrays as parameters, it is necessary **to specify the number of columns of the array**. Otherwise, we get a compilation error

```
#define ROWS 3
#define COLS 2

int readMatrix (int matrix[ROWS][COLS]);
```

The number of columns of the matrix is explicitly indicated

The number of rows of the matrix is optional (it is recommended to use it)

In the general case, it is necessary to specify the size of each dimension of the array but the first one, which is optional

Using `const` with multiple-dimension arrays is not recommended

5. Parameters – Arrays and structures

Arrays as parameters

Write a C program that calculates the largest element of a two-dimension array by using two functions: *largest*, to obtain the largest value of the matrix; *printMatrix*, to print the matrix

```
#include <stdio.h>

#define ROWS 2
#define COLS 3

int largest(int matrix[ROWS][COLS]);
void printMatrix(int matrix[ROWS][COLS]);

int main(void) {
          int a[ROWS][COLS];
          int i, j;

          /* Initialize matrix with any value */
          for (i=0; i<ROWS; i++)
                    for (j=0; j<COLS; j++)
                              a[i][j]= (i+j);

          /* Print matrix and largest value */
          printMatrix(a);
          printf("The largest value is %i\n", largest(a));

          return 0;
}
```

5. Parameters – Arrays and structures

Arrays as parameters

```
int largest(int matrix[ROWS][COLS]) {
    int i, j, max;

    max = matrix[0][0];

    for (i=0; i < ROWS; i++)
        for (j=0; j < COLS; j++)
            if (max < matrix[i][j])
                  max = matrix[i][j];

    return max;
}

void printMatrix(int matrix[ROWS][COLS]) {
    int i, j;

    for (i=0; i<ROWS; i++){
        for (j=0; j<COLS; j++)
               printf("%i\t", matrix[i][j]);

        printf ("\n");
    }

    return;
}
```

Drawbacks of passing a multiple-dimension array as parameter

- It is necessary to specify the size of all the dimensions of the array (but the first one, which is optional)
- Therefore, it is not possible to create a function able to manage multiple-dimension arrays with different size
- This problem can be solved by using dynamic memory allocation (which is briefly described in Lesson 8)

5. Parameters – Arrays and structures

Structures as parameters

A function can receive as parameters basic data types, arrays, and **structures**

By default, **structures are passed by value**. Changing a structure attribute inside the function has no effect out of the function

Structures **can be passed by reference**. In this case:

- The formal parameter must be declared as a pointer to the structure
- The actual parameter must be passed by using the & operator

To avoid compilation errors, structures must be defined before the prototypes of the functions that use them as parameters

Additionally, structures can be the return value of a function

5. Parameters – Arrays and structures

Structures as parameters

Write a C program that reads the coordinates of a point and calculates the distance to the origin (0, 0) by using three functions: *readPoint*, to read coordinates values; *distance*, to calculate the distance between two points; *getOrigin*, to return the coordinate origin point

jgromero@inf.uc3m.es

Universidad Carlos III de Madrid
www.uc3m.es

5. Parameters – Arrays and structures

Structures as parameters

ructures as arguments

```
#include <stdio.h>
#include <math.h>

struct Point {
          float x;
          float y;
};

void readPoint(struct Point *p);
float distance(struct Point p1, struct Point p2);
struct Point getOrigin(void);

int main(void) {
          struct Point point;
          struct Point origin_point;

          readPoint(&point);
          origin_point = getOrigin();

          printf("Distance: %f \n", distance(point, origin_point) );

          return 0;
}
```

5. Parameters – Arrays and structures

Structures as parameters

```
void readPoint(struct Point *p) {
        printf("x? ");
        scanf("%f", &p->x);
        printf("y? ");
        scanf("%f", &p->y);
}

float distance(struct Point p1, struct Point p2) {
        return sqrt( pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) );
}

struct Point getOrigin(void) {
        struct Point o;
        o.x = 0;
        o.y = 0;
        return o;
}
```

`p->x` is equivalent to **`(*p).x`**

Outline

1. Modular programming
2. Function declaration and definition
3. Function calling
4. Parameters – Call by value and by reference
5. Parameters – Arrays and structures
6. **Scope of variables in functions**
7. Library functions

The **scope of a variable** is the section of the code in which the variable is valid –i.e., it can be accessed and used

Local variables

- Are declared inside a function –at the beginning of a code block
- Are valid in the code block in which they are declared –if they are declared at the beginning of a function, they are valid in the whole function
- **Types**
  - *automatic* (by default)
    - Automatically created when the function starts
    - Automatically destroyed when the function finishes
  - static (we are not using them!)
    - The value is kept in different executions of a function

Remember: The **scope of the formal parameters of a function** is the function in which they are declared –they works as local variables

6. Scope of variables in functions

Local variables

```
#include <stdio.h>

void f(int y);

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

        f(a);
        printf("(Main) Local variable a: %i\n", a);
        printf("(Main) Local variable x: %i\n", x);

        return 0;
}

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

        printf("(Function) Parameter y: %i\n", y);
        printf("(Function) Local variable x: %i\n", x);

        return;
}
```

this **`x`** is local to **`main`**

this **`x`** is local to **`f`** and different to **`x`** in **`main`**

```
(Function) Parameter y: 1
(Function) Local variable x: 4
(Main) Local variable a: 1
(Main) Local variable x: 3
```

6. Scope of variables in functions

Local variables

```
#include <stdio.h>

void f(int y);

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

        f(a);
        printf("(Main) Local variable a: %i\n", a);
        printf("(Main) Local variable x: %i\n", x);

        return 0;
}

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

        printf("(Function) Parameter x: %i\n", x);
        printf("(Function) Local variable x: %i\n", x);

        return;
}
```

Compilation error: same name for variable and parameter

6. Scope of variables in functions

Local variables

```
#include <stdio.h>

void f(int y);

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

        f(a);
        printf("(Main) Local variable a: %i\n", a);
        printf("(Main) Local variable x: %i\n", x);

        return 0;
}

void f(int a) {
        int x = 4;
        a = 10;
        printf("(Function) Parameter a: %i\n", a);
        printf("(Function) Local variable x: %i\n", x);
}
```

this variable **a** is local to **main**

a copy of **a** value is passed to **f**

this parameter **a** is local to **f**

```
(Function) Parameter a: 10
(Function) Local variable x: 4
(Main) Local variable a: 1
(Main) Local variable x: 3
```

Global variables

Declared out of the user-defined functions (and out of the main function)

Can be accessed from any function

Global variables are evil!

> Do not declare global variables

> Do not use global variables inside functions

> To exchange information between functions, values are passed as parameters, in order to make programs more readable, easier to understand, and easier to debug

6. Scope of variables in functions

Global variables example

```
#include <stdio.h>

int a;

void f(void);

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

        f();
        printf("(Main) Global variable a: %i\n", a);
        printf("(Main) Local variable x: %i\n", x);

        return 0;
}

void f(void) {
        int x = 3;
        a = 10;
        printf("(Function) Global variable a: %i\n", a);
        printf("(Function) Local variable x: %i\n", x);
}
```

declaration of global variable **a**
NOT RECOMMENDED

assigning a value to global variable **a**

assigning a value to global variable **a**

```
(Function) Global variable a: 10
(Function) Local variable x: 3
(Main) Global variable a: 10
(Main) Local variable x: 1
```

Outline

1. Modular programming
2. Function declaration and definition
3. Function calling
4. Parameters – Call by value and by reference
5. Parameters – Arrays and structures
6. Scope of variables in functions
7. **Library functions**

7. Library functions

Use

- C language provides **several standard libraries with functions** implementing common tasks that can be used by the programmer
- C standard functions **can be called from any user-defined function** (including the main function) to perform calculations
- C standard functions are organized in libraries
  - All the functions of a library are declared in the same header file (.h)
    - Example: basic I/O operations (`printf`, `scanf`, etc.) are declared in the file `stdio.h`
  - A program can include as many header files as necessary
    - Example: `#include <stdio.h>`

jgromero@inf.uc3m.es

Universidad Carlos III de Madrid
www.uc3m.es

7. Library functions
Common standard libraries

`<complex.h>`	Complex numbers operations
`<ctype.h>`	Character management
`<errno.h>`	Error control
`<float.h>`	Additional functionalities for floats
`<math.h>`	Mathematical functions
`<stdio.h>`	Basic Input/Output operations
`<stdlib.h>`	Absolute value, random number generation, search and sort, string conversion, memory management, and communication with the OS
`<string.h>`	String management
`<time.h>`	Time and date functions

(see Kernighan & Ritchie, “The C Programming Language”, Appendix B)

7. Library 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
acos(x)	double	$Cosin^{-1}$ of **x**	math.h
printf(..)	int	Print data on the screen	stdio.h
scanf(..)	int	Read data from the keyboard	stdio.h
strcpy(s1,s2)	char*	Copies string **s2** into string **s1**	string.h
strlen(s1)	int	Number of characters of **s1**	string.h
strcmp(s1, s2)	int	Compares **s1** and **s2**; if equal, it returns 0	string.h

7. Library functions
Programmer defined libraries

Programmers needs to include the .h file of the library that is used in the program

In general, the main function and the additional functions can be stored in **separated files**

Similar functions are grouped into the same file to implement a programmer-defined library

Two files are created:

- **.h** >> *function prototypes, constants and* `struct` *types*
- **.c** >> *function implementation*

The file that uses an external function must import the *.h* prototypes file

functions.h

Header file

```c
/*! Read values from the keyboard to store them into array a
    @param[out] a Array to be read [is modified]
    @param[in]  n Maximum size of the array
*/
void read_array(int a[], int n);

/*! Print array values
    @param[in] a Array to print
    @param[in] n Size of the array
*/
void print_array(int a[], int n);
```

functions.c

Source file

```c
#include <stdio.h>

void read_array(int a[], int n) {
    int i;
    printf("Please enter %i array values \n", n);

    for(i=0; i<n; i++)
        scanf("%i", &a[i]);

    return;
}

void print_array(int a[], int n) {
    int i;
    printf("Printing %i array values \n", n);

    for(i=0; i<n; i++)
        printf("%i ", a[i]);

    return;
}
```

7. Library functions

Programmer defined libraries

```
#include <stdio.h>
#include "functions.h"

#define SIZE 10

int main(void) {
    int array[SIZE];

    read_array(array, SIZE);
    print_array(array, SIZE);

    return 0;
}
```

main.c
Main file Uses external functions

7. Library functions

Programmer defined libraries in Dev-Cpp

1. Create a New project
2. Add files to Project
   Project > New file (if the file has not been previously created)
   Two files must be created (one for *.h*, one for *.c*)
   Project > Add to project (if the file has been previously created)
3. Select the files of the module
   .c and *.h* files
   These files must be stored in the same folder as the *main* file
4. Implement the *main* function in *main.c* (no *.h* for this file)
   Use the directive `#include "file.h"` to import the functions
5. Compile
   Execute > Compile only compiles files modified since the last compilation
   Execute > Compile current file compiles the file active in the IDE
   Execute > Rebuild all recompiles all the modules of the project

Summary

Remarks

- A function is an independent piece of code aimed to solve a concrete task. Functions return a single value (at most, `void` can be used)
- Functions must be declared (*prototype*) and defined (*implementation*). The prototype and the implementation can be in different files (.h and .c, respectively)
- The prototype of a function includes the name of the function, the returning type, and the type and the name of each parameter. The declaration of the function prototype ends with *;*
- Formal parameters are the parameters as declared in the function prototype and used in the function definition
- A function with no parameters uses `void` in the parameter list
- A function ends when the corresponding `return` instruction is executed
- `return` must be always used inside functions (although it can be omitted in functions returning `void`). When `return` is executed, a result value is given back to the calling function

Summary

Remarks

- To call a function, the name of the function and the list of actual parameters (that is, argument values) are used
- An actual parameter passed by value is not changed inside the function
- An actual parameter passed by reference can be changed inside the function. Passing parameters by reference involve the use of pointers
  - The actual parameter is a memory address (`&`)
  - The formal parameter is a pointer (`*`)
- A local variable can be only accessed from the function in which it is declared (actually, only from the block in which it is declared)
- Global variables must not be used. To pass values between functions, parameters must be used

- To pass one-dimension arrays as parameters, the size must be specified as a second parameter

```
int addData(int vector[], int n);
```

- To pass multiple-dimension arrays as parameters, the size of the dimensions (but the first one) must be specified

```
int largest(int matrix[][COLS]);
```

- When an array is passed as an actual parameter to a function, the address of the first element of the array is being passed indeed
  - Inside a function, the values of the elements of the array can be changed

```
int a[SIZE];
readArray(a, SIZE);
```

Bibliography

Recommended lectures

Basic

- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapter 8
- Stephen Prata. *C Primer Plus.* Sams, 2004 (5th Edition) – Chapter 9 (everything but *Recursion*)
- Stephen G. Kochan. *Programming in C.* Sams, 2004 (3rd Edition), Programming in C – Chapter 8.6 (*Functions and arrays*), 9.2 (*Functions and structures)*

Additional information

- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapter 9.9 (*Designing a program)*

Outline

1. **Modular programming**
2. **Function declaration and definition**
3. **Function calling**
4. **Arguments – Call by value and by reference**
5. **Arguments – Arrays and structures**
6. **Scope of variables in functions**
7. **Library functions**