PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

FUNCTIONS

INTRODUCTION AND MAIN

- All the instructions of a C program are contained in *functions*.

  - ✓ C is a procedural language
  - ✓ Each function performs a certain task

- A special function name is **main( )**: the function with this name is the first one to run when the program starts.

- The first command executed in your program is the first command in main (top to bottom)

- All other functions are subroutines of the **main( )** function can have any names you wish.

- Every function is defined exactly once. A program can declare and call a function as many times as necessary.

FUNCTION DEFINITION

- The *definition* of a function consists of a *function head* (or the *declarator*), and a *function block*.

- The function head specifies the name of the function, the type of its return value, and the types and names of its parameters, if any.

- The statements in the function block specify what the function does.

```
type function_name (type parameter_name1, type parameter_name2,…) {

    //statements;
}
```

FUNCTION RETURN

- Always write the return-value type
  - ✓It is **int** if not specified

FUNCTION DECLARATION

- By declaring a function before using it, you inform the compiler of its type: in other words, a *declaration* describes a function's interface

- A declaration defines the **type** of a function

  ✓ The number and types of its parameters
  ✓ The type of what returned

- The identifiers of the parameters in a function declaration are optional. If you include the names, their scope ends with the prototype itself

type function_name (type parameter_name1, type parameter_name2,…);

type function_name (type, type,…);

Both valid declarations

IDENTIFIER SCOPE 2

- *Block scope:* identifiers declared within a block have block scope. You can use such an identifier only from its declaration to the end of the smallest block containing that declaration. The smallest containing block is often, but not necessarily, the body of a function definition. In C11, **declarations do not have to be placed before all statements in a function block**. The parameter names in **the head of a function definition also have block scope**, and are valid within the corresponding function block.

- *Function prototype scope:* The parameter names in a function prototype have function prototype scope. Because these parameter names are not significant outside the prototype itself, they are meaningful only as comments, and can also be omitted.

EXAMPLE 1

```
int p;

int fun1 (int q);

int fun2 (int r)
{
    int s = r + 3;
    while( r != s)
       r--;
}
```

Function prototype scope

File scope   Block scope

p, fun1, fun2   q   r,s
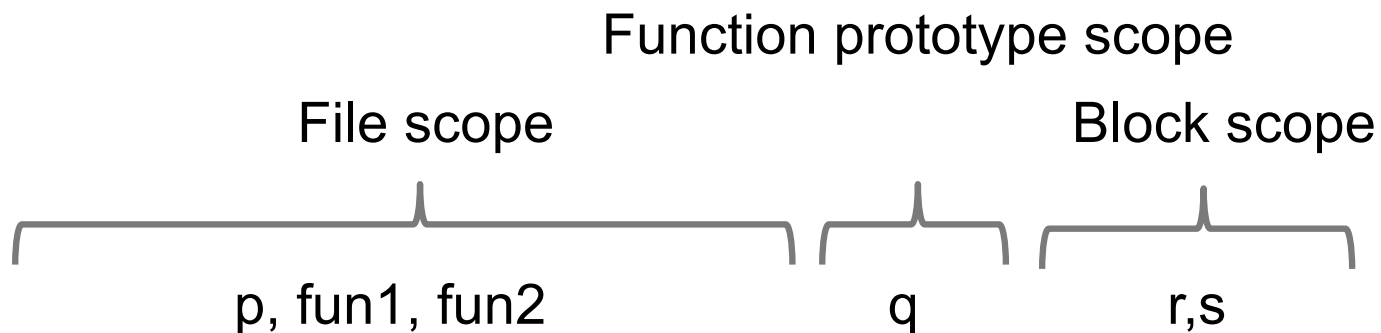
FUNCTION PARAMETERS

- The parameters of a function are ordinary local variables

- The program creates them, and initializes them with the values of the corresponding arguments, when a function call occurs.

- Their scope is the function block (**block scope**).

```
long double factorial( unsigned int n )
{
  long double f = 1;
  while ( n > 1 )
    f *= n--;
  return f;
}
```

DECLARATION AND DEFINITION

A function definition is also a function declaration

```
int fun2 (int);

int main() {
    int a= 5;
    a= fun2(a);
}

int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
return r;
}
```

=

```
int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
    return r;
}

int main() {
    int a= 5;
    //commands
    a= fun2(a);
}
```

so

- Before calling (using) a function you need to declare it
  - ✓You can declare all the functions at the top of the file, and define them in any order below in the same file
  - ✓Otherwise you can avoid declaring function at the top, but you need to define them before they the are called

- A function needs also to be defined, otherwise you do not know hat to do when it is called

- In the same file, you cannot have two definitions of functions with the same header and identifier (name)

EXAMPLE

```
int fun1 (int);
int fun2 (int);

int main() {
    int a= 5;
    a= fun1(a);
}

int fun1(int q) {
    return (fun2(q)+1);
}

int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
    return r;
}
```

```
int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
    return r;
}

int fun1(int q) {
    return (fun2(q)+1);
}

int main() {
    int a= 5;
    a= fun1(a);
}
```

EXAMPLE

```
int fun1 (int);

int main() {
    int a= 5;
    a= fun1(a);
}

int fun1 (int q) {
    return (fun2(q)+1);
}

int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
    return r;
}
```

```
int fun1 (int);
int fun2 (int);

int main() {
    int a= 5;
    a= fun1(a);
}

int fun1 (int q) {
    return (fun2(q)+1);
}

int fun2 (int r) {
    int s = r + 5;
    while( r != s)
        r--;
    return r;
}
```

RETURN

- The return statement ends execution of the current function, and jumps back to where the function was called:

    return [expression];

- *expression* is evaluated and the result is given to the caller as the value of the function call.

- This *return value* is converted to the function's return type, if necessary.

- A function can contain any number of return statements

  ✓The first return executed exits from a function

- Return is an unconditional jump statement (as **goto**)

EXAMPLE

```
// Return the smaller of two integer arguments.
int min( int a, int b )
{
    if ( a < b )
        return a;
    else
        return b;
}

int main()                                    5
{
    int a= 5;
    int b= 6;
    int c= min(a ,b);
    printf("%d", c);
}
```

RETURN (2)

- A return statement with no *expression* can only be used in a function of type void. In fact, such functions do not need to have a return statement at all

- If no return statement is encountered in a function, the program flow returns to the caller when the end of the function block is reached

MAIN

MAIN() FUNCTION

- You can define the main( ) function in one of the following two forms:

- int main(void) { /* ... */ }
  A function with no parameters, returning int

- int main(int argc, char *argv[]) { /* ... */ }
  A function with two parameters whose types are int and char **, returning int

PARAMETERS OF MAIN()

- The parameters argc and argv (which you may give other names if you wish) represent your program's command-line arguments. This is how they work:

- **argc** (short for "argument count") is either 0 or the number of string tokens in the command line that started the program. The name of the program itself is included in this count.

- **argv** (short for "arguments vector") is an array of pointers to char that point to the individual string tokens received on the command line:

    ✓The number of elements in this array is one more than the value of argc; the last element, argv[argc], is always a null pointer.

    ✓The first string, argv[0], contains the name by which the program was invoked. If the execution environment does not supply the program name, the string is empty.

    ✓If argc is greater than 1, then the strings argv[1] through argv[argc - 1] contain the program's command line arguments.

A CLASSICAL EXAMPLE

```c
#include <stdio.h>
int main( int argc, char *argv[] )
{
  if ( argc == 0 )
    puts( "No command line available." );
  else {

    printf( "The program now running: %s\n", argv[0] ); // Print the name of the
                                                         // program.
    if ( argc == 1 )
      puts( "No arguments received on the command line." );

    else {
      puts( "The command line arguments:" );
      for ( int i = 1; i < argc; ++i )                   // Print each argument on
                                                         // a separate line.

        puts( argv[i] );
    }
  }
}
```

OUTPUT FOR THE EXAMPLE

- Suppose we run the program on a Unix system by entering the following command line:

  ✓ $ **./args one two "and three"**

- The output is then as follows:

```
The program now running: ./args
The command line arguments:
one
two
and three
```

RETURN OF MAIN

- The value returned by main is a value that is passed to the parent process that executed it (e.g., the shell)

- The return value for main should indicate how the program exited. Normal exit is generally represented by a 0 return value from main.

- Abnormal termination is usually signalled by a non-zero return

HOW FUNCTIONS ARE EXECUTED

RETURNING FROM A FUNCTION

- If the program reaches a return statement or the closing brace **}** of the function block, execution of the function ends, and the program jumps back to the calling function.

- If the program "falls off the end" of the function by reaching the closing brace, the value returned to the caller is undefined.

- For this reason, **you must use a return statement** to stop any function that does not have the type void. The value of the return expression is returned to the calling function.

CALL A FUNCTION

- The instruction to execute a function—the function call—consists of the function's name and the operator **( )**

- For example, the following statement calls the function **maximum( )** to compute the maximum of two numbers

  ✓maximum(a, b);

- The program first allocates storage space for the parameters, then copies the argument values to the corresponding locations.

- Then the program jumps to the beginning of the function, and execution of the function begins with first variable definition or statement in the function block.
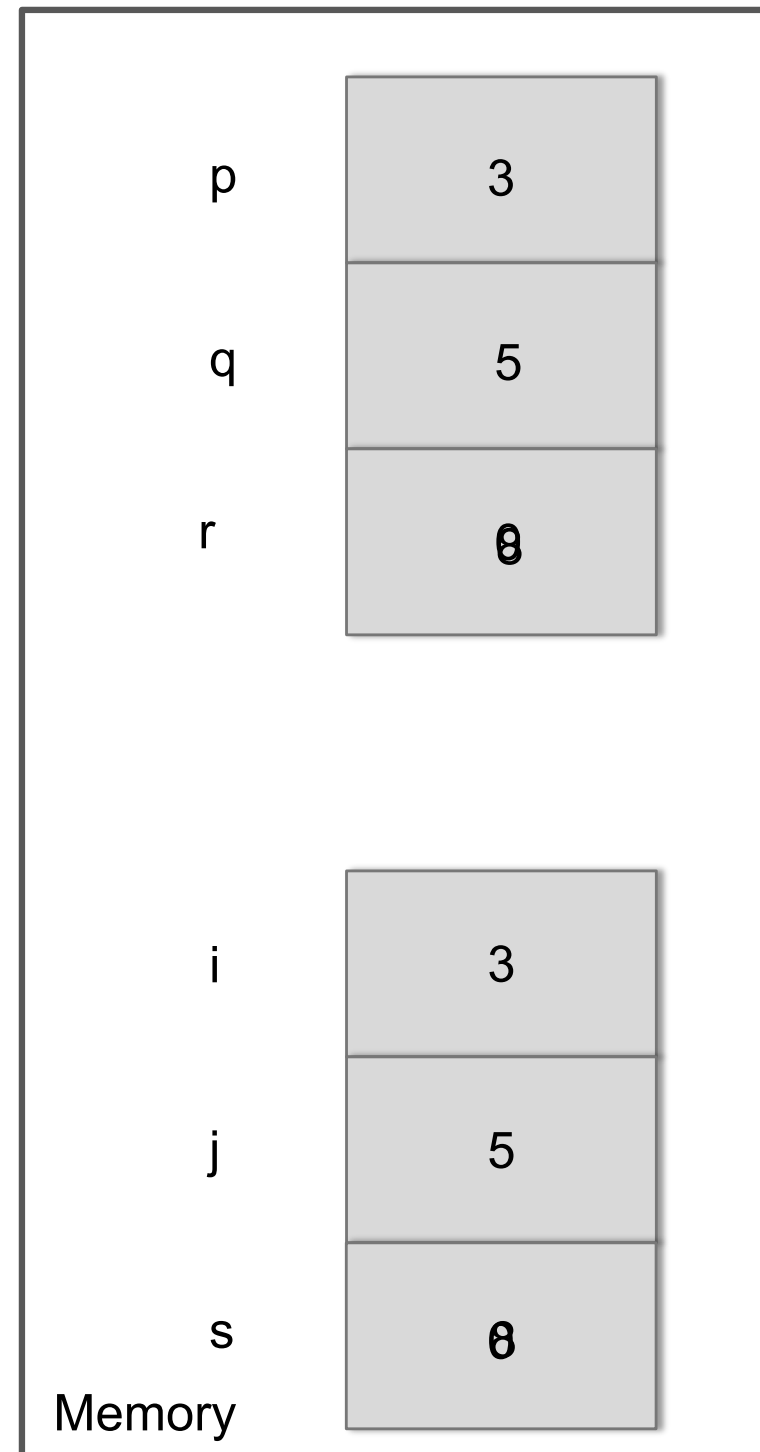
EXAMPLE

i and j are passed by value!

```c
#include <stdio.h>


int sum(int p, int q) {
    int r= 0;
    r= p + q;
    return r;
}

int main( void ) {
    int i =3, j =5, int s= 0;
    s= sum(i, j);
    printf("%d\n", s);
    return 0;
}
```

8

p	3
q	5
r	0

i	3
j	5
s	0

Memory

FORMAL AND ACTUAL PARAMETERS

- When a function is *called*, the values (expressions) that are passed in the call are called the *arguments* or *actual parameters* (both terms mean the same thing).

- At the time of the call each actual parameter is assigned to the corresponding *formal* parameter in the function definition.

- p and q are formal parameters in the example before, the value of i and j are the actual parameters.

VARIABLE ALLOCATION

- Memory allocation means that part of the memory is reserved, for instance for a variable

- Automatic variable is a local variable which is allocated and deallocated automatically when program flow enters and leaves the variable's scope

- By memory deallocation we mean that part of the memory is no longer reserved, for that variable

- Variables p and q are allocated when the function is called, deallocated when we exit from the function

OVERLOADING OF FUNCTIONS

NOT POSSIBLE IN C!

- Function overloading is a feature of a programming language that allows one to have many functions with same name but with different signatures (parameters' type and return type)

  - ✓ This feature is present in most of the Object Oriented Languages such as C++ and Java
  - ✓ But C doesn't support this feature

- To overload a function, in C++ you can define it several times, each uniquely identified by the type of the arguments it accepts; return type is not considered

EXAMPLE

```c
#include<stdio.h>

int min(int x, int y) {
    return ( x < y  ? x : y);}

float min(float x, float y) {
    return ( x < y  ? x : y);
}

int main( ) {
    int a= min(4, 1);
    float b= min(4.56F, 1.23F);
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc esempio.c
esempio.c:8:7: error: conflicting types for 'min'
float min(float a, float b) {
     ^

MacBook-Francesco:ProgrammI francescosantini$ g++ esempio.c
```

SU LIBRO

- Sezioni 5.1-5.6
- Sezione 15.3
- Sezione 7.4