Introdução à Programação 2020/2021

António J. R. Neves
Daniel Corujo

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

an@ua.pt / dcorujo@ua.pt

http://elearning.ua.pt/

Topics

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

- Functions
- Functions parameters
- Arrays as parameters
- Recursive function calls

- All the instructions of a C++ program are contained in one or more functions.
- Each function performs a certain task.
- **`main()`** is a special function: it is the entry point of the program (this function is implicitly called when the program starts).

```
int main(void)    // function definition
{
   double x = 9.0;
   double y = sqrt(x);  // example of a function call
   double z = sqrt(2.0 * x); // another function call
   return 0;
}
```

- Every function is defined exactly once. A program can call a function as many times as necessary (**`sqrt`** in the previous example).

- The definition of a function consists of a function head 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 name( parameter_declarations )   Function head
{
      /* declarations, statements */  Function block
}
```

- The return type may be **`void`** or any object type except arrays or functions. However it can return a pointer to a function or a pointer to an array.

- The parameters declaration is a comma-separated list of variable declarations.
- If the function has no parameters, this list is either empty or contains merely the word **`void`**.
- Example:

```
double cylinderVolume( double r, double h )
{
  const double pi = 3.14159265358979324;
  return pi * r * r * h;
}
```

- This function has the name **`cylinderVolume`**, and has two parameters, **`r`** and **`h`**, both with type **`double`**. It returns a value of type **`double`**.

- A function can also be declared without being defined; it must still be defined elsewhere, perhaps later in the file.
- Header files, such as **`iostream.h`** and **`cmath.h`**, declare several functions or objects, such as **`cout`**, **`cin`**, **`sqrt`**, ...
- The declaration only is a so-called function prototype. It is a copy of the function head, followed by a semicolon (;).
- Example:

```
double cylinderVolume( double r, double h );
```

- When a function is used in a source code file before it is defined it is necessary to provide a prototype of the function before it is first used; it must match the function head of the actual definition.

- The parameters of a function are ordinary local variables.
- They are created and initialized with the values of the corresponding function arguments when a function call occurs (this is called “call by value”).
- Their scope is the function block.
- A function can change the value of a parameter without affecting the value of the argument in the context of the function call.

```
double factorial( unsigned int n )
{
    double f = 1.0;
    while(n > 1)
        f *= (double)(n--);
    return f;
}
```

- **`factorial`** modifies its (local copy of) parameter **`n`**.

- The following function swaps the values of its arguments (**but only inside the function**).

```
void bad_swap( int x, int y )
{
    printf("x=%d y=%d\n", x, y);
    int tmp = x;
    x = y;
    y = tmp;
    printf("x=%d y=%d\n", x, y);
}
```

- if `i = 3` and `j = 7` then **`bad_swap(i,j)`** prints

```
x=3 y=7

x=7 y=3
```

After the **`bad_swap`** function call the value of `i` is still 3 and the value of `j` is still 7.

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

- Using pointers the situation is different:

```
void good_swap( int *xPtr, int *yPtr )
{
  cout << "x= " << *xPtr << " y= " << *yPtr << endl;
  int tmp = *xPtr;
  *xPtr = *yPtr;
  *yPtr = tmp;
  cout << "x= " << *xPtr << " y= " << *yPtr << endl;
}
```

- if `i = 3` and `j = 7` then `good_swap(&i,&j)` prints

```
x= 3 y= 7
x= 7 y= 3
```

just like **`bad_swap`** did. After the **`good_swap`** function call, however, the value of `i` is now 7 and the value of `j` is now 3 (the memory regions pointed to by the two pointers were swapped, the pointers themselves were not changed).

- To declare an array as an argument to a function we can use the following forms:

  `type name[]` or **`type *name`**

- Array names are pointers to their first elements.

```
int sumArray( int *a, int len ) {
    int sum = 0;
    for( int i = 0 ; i < len ; i++ )
        sum += a[i];
    return sum; }

int main( void ) {
    const int DIM = 5;
    int x[DIM] = {100, 200, 300, 400, 500};
    cout << sumArray(x, DIM) << endl; // x is the same as
    return 0; }                       //   &x[0]
```

- Let us write a function to read an array of integers
- First, we need to decide:
  - Its name (say, **`read_int_array`**)
  - What it returns (say, the number of integers placed in the array)
  - Its arguments (say, one argument holding a pointer to the beginning of the array and another argument holding the maximum number of elements that can be placed in the array)
  - Its behaviour (we will ask first for the number of integers to input)
- From these specifications our function should look like this:

```
int read_int_array( int a[], int max_n );
```

- Now, the actual code:

```
int read_int_array( int a[], int max_n )
{
  int n;
  cout << "n = ";
  cin >> n;

  for(int i = 0; i < n; i++ )
  {
    cout << "a[" << i << "] = ";
    cin >> a[i];
  }

  return n; // success (note that you should implement a more robust function)
}
```

Example (read an array)

- Now we use the **`read_int_array`** function to read two arrays (the **`for`** cycle prints the sum of the two arrays, element by element)

```
int main(void)
{
  int n1, a1[10], n2, a2[10];

  cout << "Introduza o primeiro array" << endl;
  n1 = read_int_array(a1, 10);
  cout << "Introduza o segundo array" << endl;
  n2 = read_int_array(a2, 10);
  if(n1 == n2 && n1 >= 1)
    for(int i = 0; i < n1; i++ )
      cout << "a1[i] + a2[i] = " << (a1[i] + a2[i]) << endl;
  return 0;
}
```

- A function may call itself (if so, it is a recursive function).
- A recursive function cannot call itself for ever (stack overflow); it must contain a test to stop the recursion.
- An example:

```
double factorial(unsigned int n)
{
  if(n < 2)
    return 1.0; // 0! and 1! are equal to 1.0
  return (double)n * factorial(n - 1);
}
```

- `factorial(3)` calls
  `factorial(2)`, which in turn calls
  `factorial(1)`, which
  does not call `factorial` anymore

- What is happening with **`factorial(3)`**:

```
double factorial(unsigned int n)
{
  cout << "Evaluating factorial " << n << " ..." << endl;
  double r;
  if(n < 2)
    r = 1.0; // 0! and 1! are equal to 1.0
  else
    r = (double)n * factorial(n - 1);
  cout << "Returning " << r << " for n = " << n << endl;
  return r;
}
```

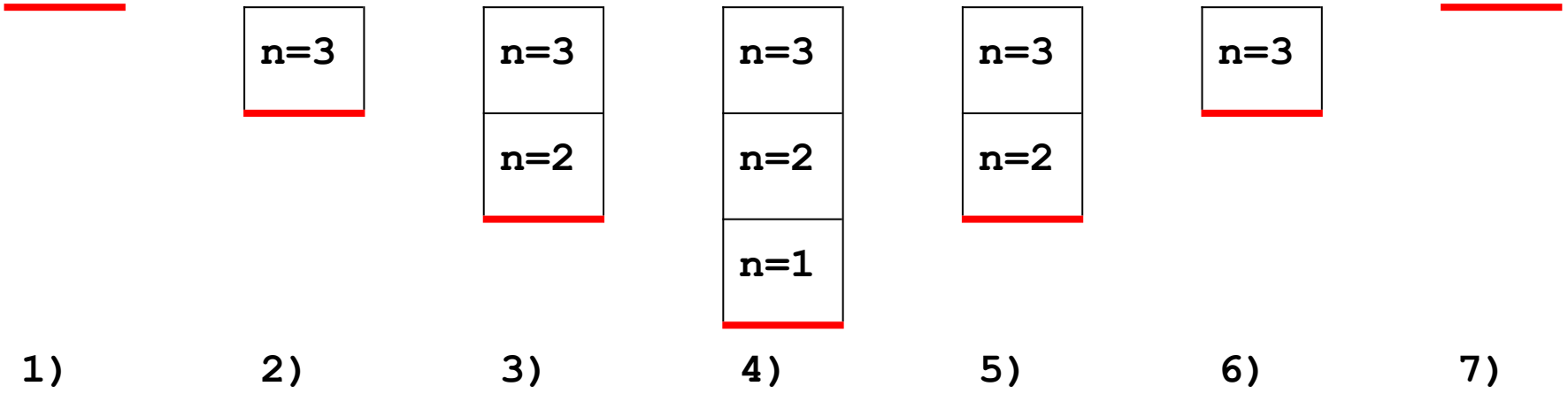
```
Evaluating factorial 3 ...
Evaluating factorial 2 ...
Evaluating factorial 1 ...
Returning 1.0 for n = 1
Returning 2.0 for n = 2
Returning 6.0 for n = 3
```

- All programs have a special memory area, called the stack.
- The stack is used to store the values of all arguments and local variables of the functions that are called.
- In most computer architectures, the stack grows downwards (towards lower addresses).
- The processor has a special register (the stack pointer), that points to the last memory position occupied by the stack.
- When a function is called, the stack pointer is decreased to reserve memory to store the value of all its arguments.
- Each argument (in general, an expression), is evaluated and its value is placed in its position in the stack.
- Automatic variables (of the function), are also placed in the stack.
- When the functions returns, the stack pointer in increased, to free the memory used by its arguments and automatic variables.

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

- The following image illustrates the stack contents while factorial(3) is being evaluated (time evolves from the left to the right, the thick red line represents the stack pointer):

1. initial stack state
2. just after the call to **`factorial(3)`**
3. just after the call to **`factorial(2)`**
4. just after the call to **`factorial(1)`**
5. just after the return from **`factorial(1)`**
6. just after the return from **`factorial(2)`**
7. just after the return from **`factorial(3)`** --- final stack state



- An overloaded function appears to perform different activities depending on the kind of data sent to it
- The compiler uses the function signature - the number of arguments, and their data types- to distinguish one function from another
- Example:
  - `void repchar();`
  - `void repchar(char);`
  - `void repchar(char, int);`
- Which one of these functions will be called depends on the number of arguments supplied in the call
- The compiler can also distinguish between overloaded functions with the same number of arguments, provided their type is different