C Developer

Functions

SUPINFO

Course Objectives

- ✓ Know how to declare a function
- ✓ Understand how to simulate a procedure
- ✓ Develop good programming habits by breaking down projects

Course Plan

1. Function Declaration
2. Splitting a Program into Modules
3. Function Parameters

1. Function Declaration

1. Function Declaration

Definition of a function

- A function is basically a **subroutine** that takes **parameters** as input and **returns** a value

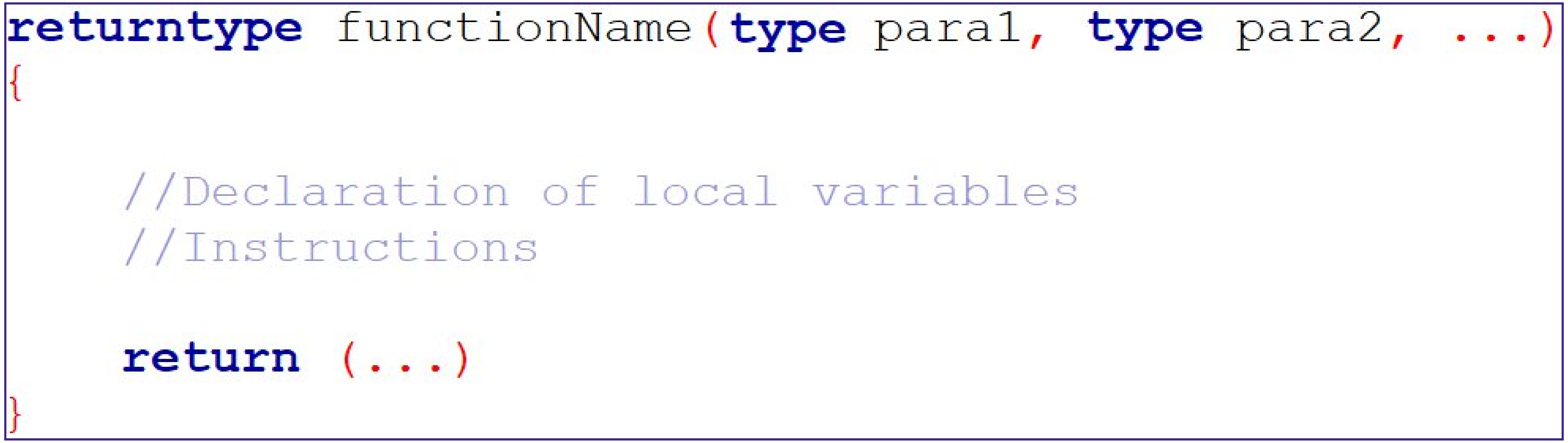
- There are no procedures in C; however, we will see how to simulate a procedure using functions that do not return any value, and how to modify the value of the parameters using pointers

- By being able to modify the value of the parameters, we will no longer be limited to input parameters, and we will be able to have output parameters or input/output parameters

1. Function Declaration

Definition of a function

- Syntax:



```
returntype functionName(type para1, type para2, ...)
{

    //Declaration of local variables
    //Instructions

    return (...)
}
```

1. Function Declaration

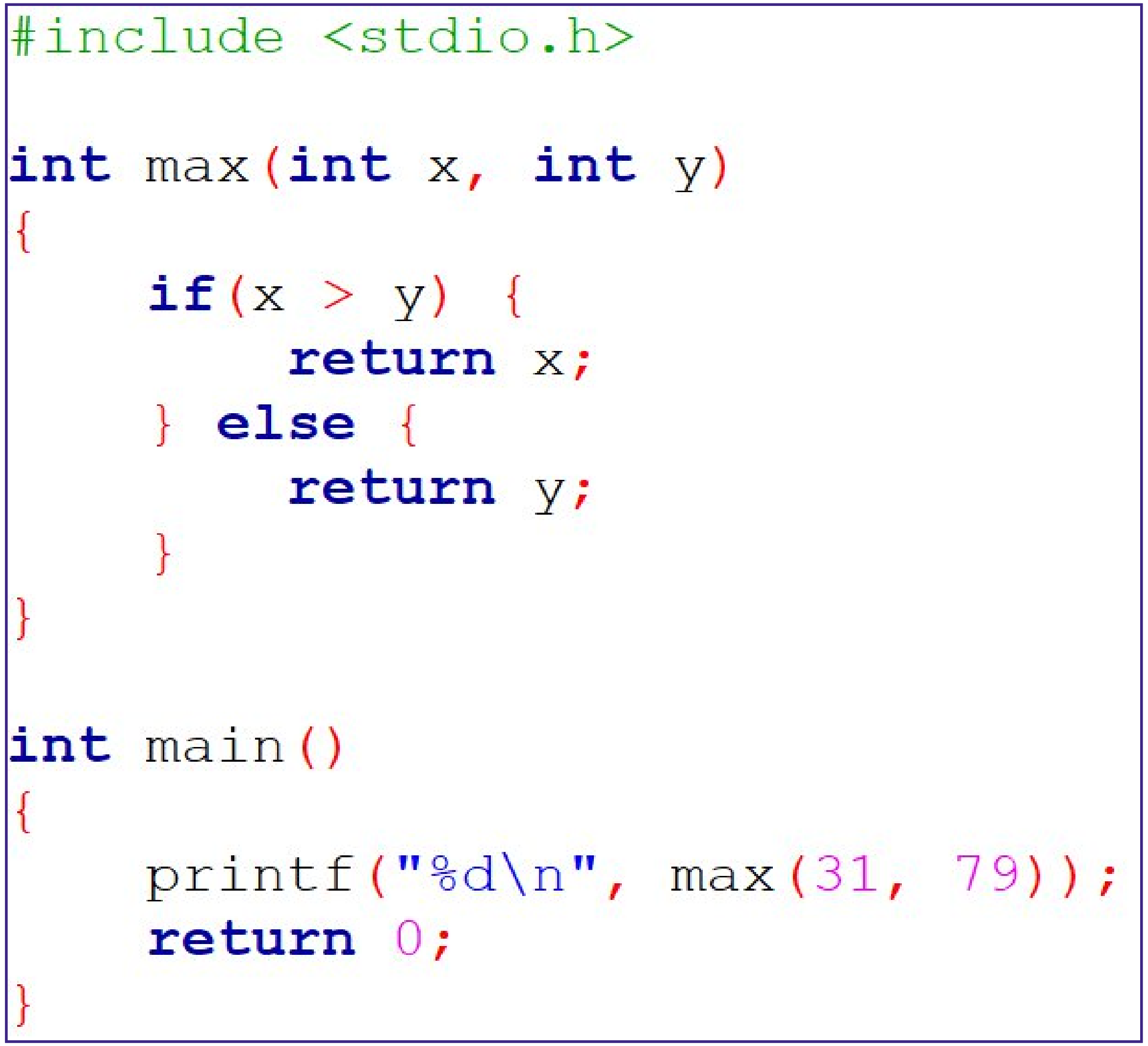
Definition of a function

- We call a function by specifying its name, and between parentheses the values we want to assign to the parameters

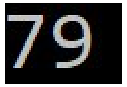
- The value returned by the function will either be assigned to a variable or displayed

1. Function Declaration

Definition of a function



```c
#include <stdio.h>

int max(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
}

int main()
{
    printf("%d\n", max(31, 79));
    return 0;
}
```



```
79
```

1. Function Declaration

Definition of a function

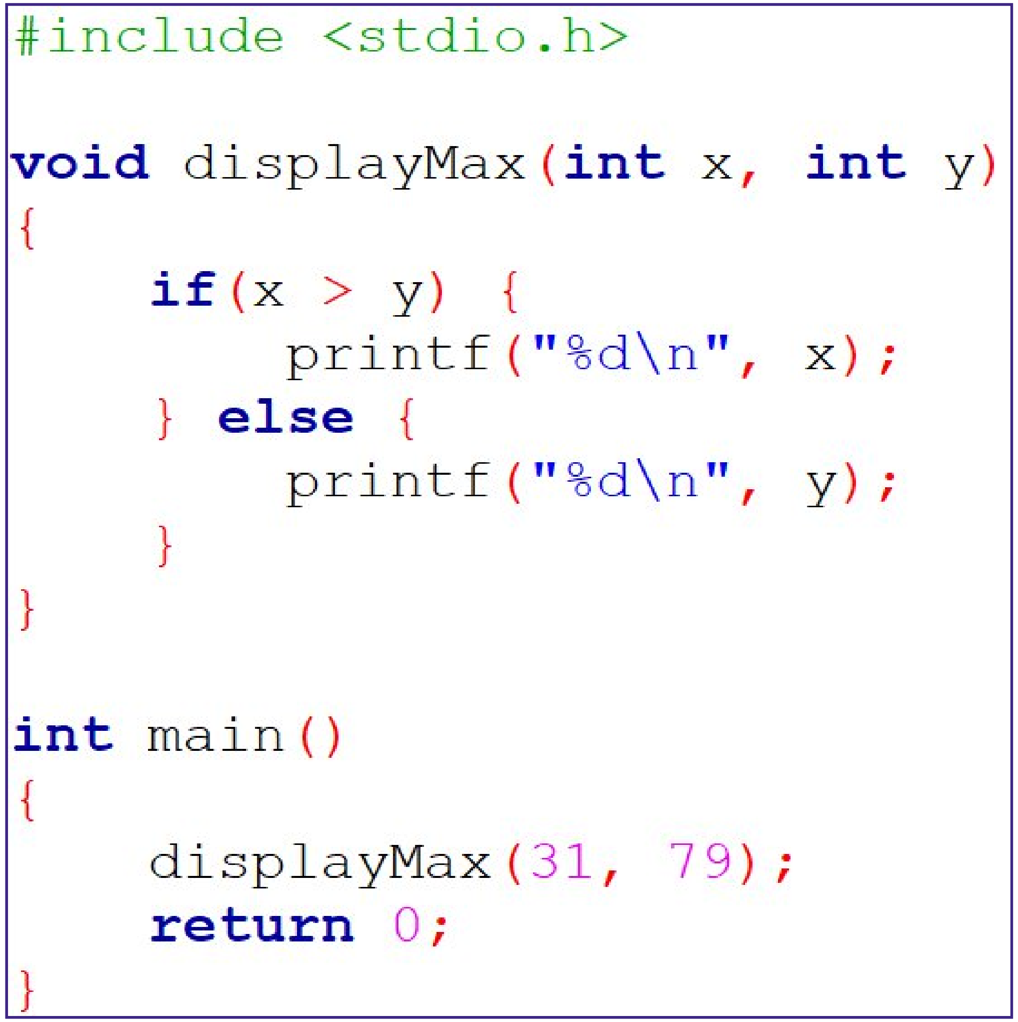
- If the function we want to create is not intended to return a value, we replace its return type by the **void** keyword

- The function will of course not contain the **return** command

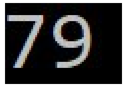
- This thus simulates a **procedure** performing an **action**

1. Function Declaration

Definition of a function



```
#include <stdio.h>

void displayMax(int x, int y)
{
    if(x > y) {
        printf("%d\n", x);
    } else {
        printf("%d\n", y);
    }
}

int main()
{
    displayMax(31, 79);
    return 0;
}
```



```
79
```

1. Function Declaration

Definition of a function

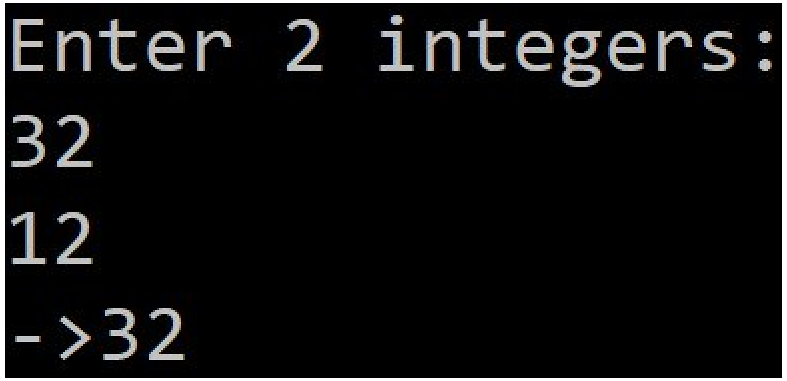
- It is possible to write functions that take no parameters as input: to indicate this, the declaration of parameters is replaced by the **void** keyword

- The function call will then simply be without content between the parentheses

1. Function Declaration

Definition of a function

```c
#include <stdio.h>

int max(void)
{
    int x, y;
    printf("Enter 2 integers:\n");
    scanf("%d%d", &x, &y);
    if(x > y) {
        return x;
    } else {
        return y;
    }
}

int main()
{
    printf("->%d\n", max());
    return 0;
}
```

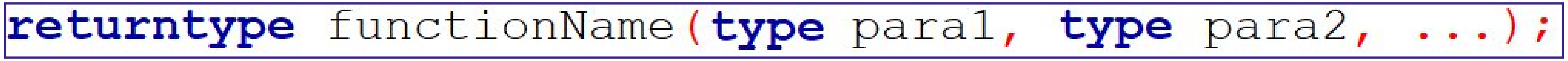


```
Enter 2 integers:
32
12
->32
```

1. Function Declaration

Function prototype

- It is a simple line indicating the type returned by the function and the types of the parameters:

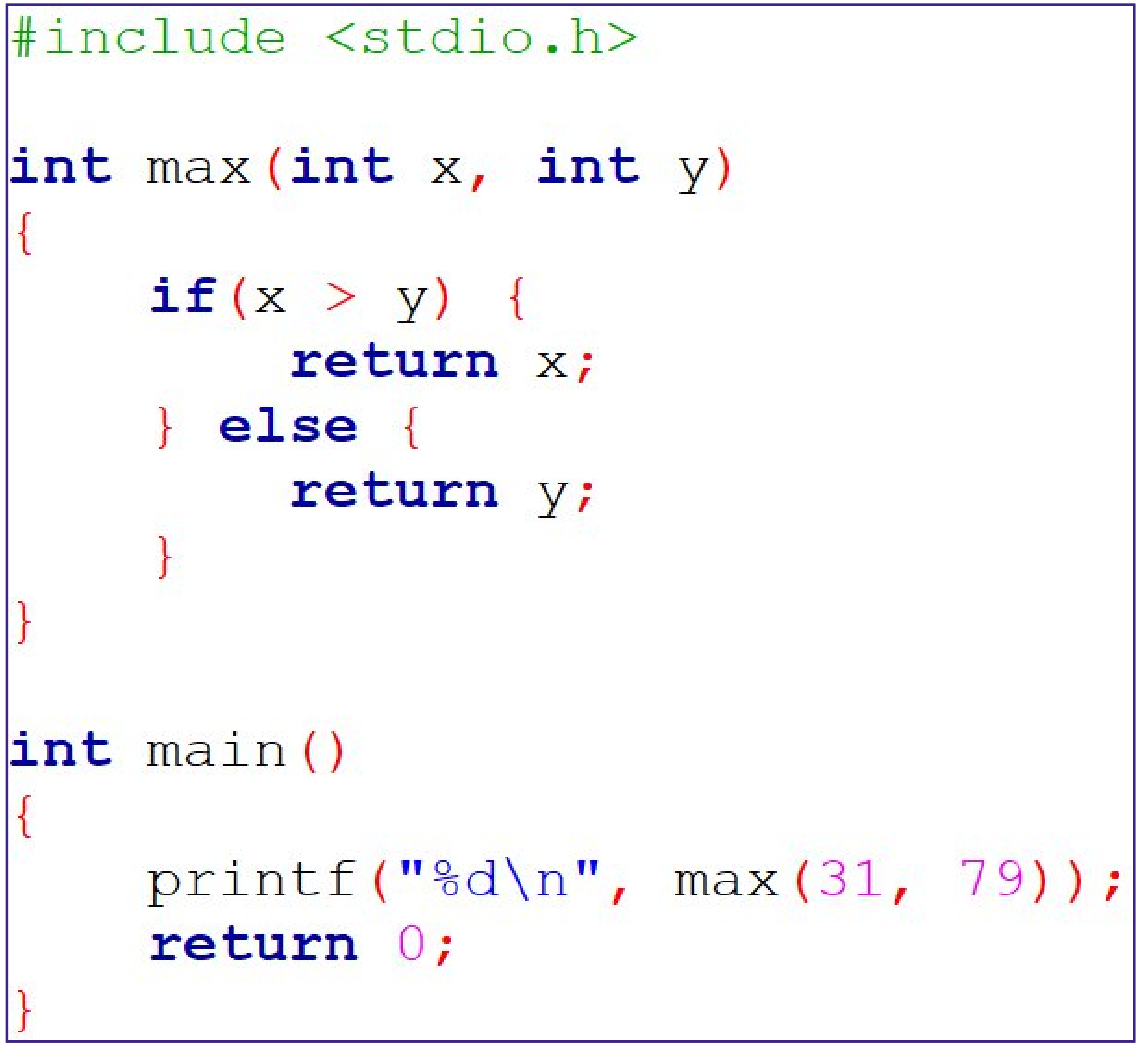


```
returntype functionName(type para1, type para2, ...);
```

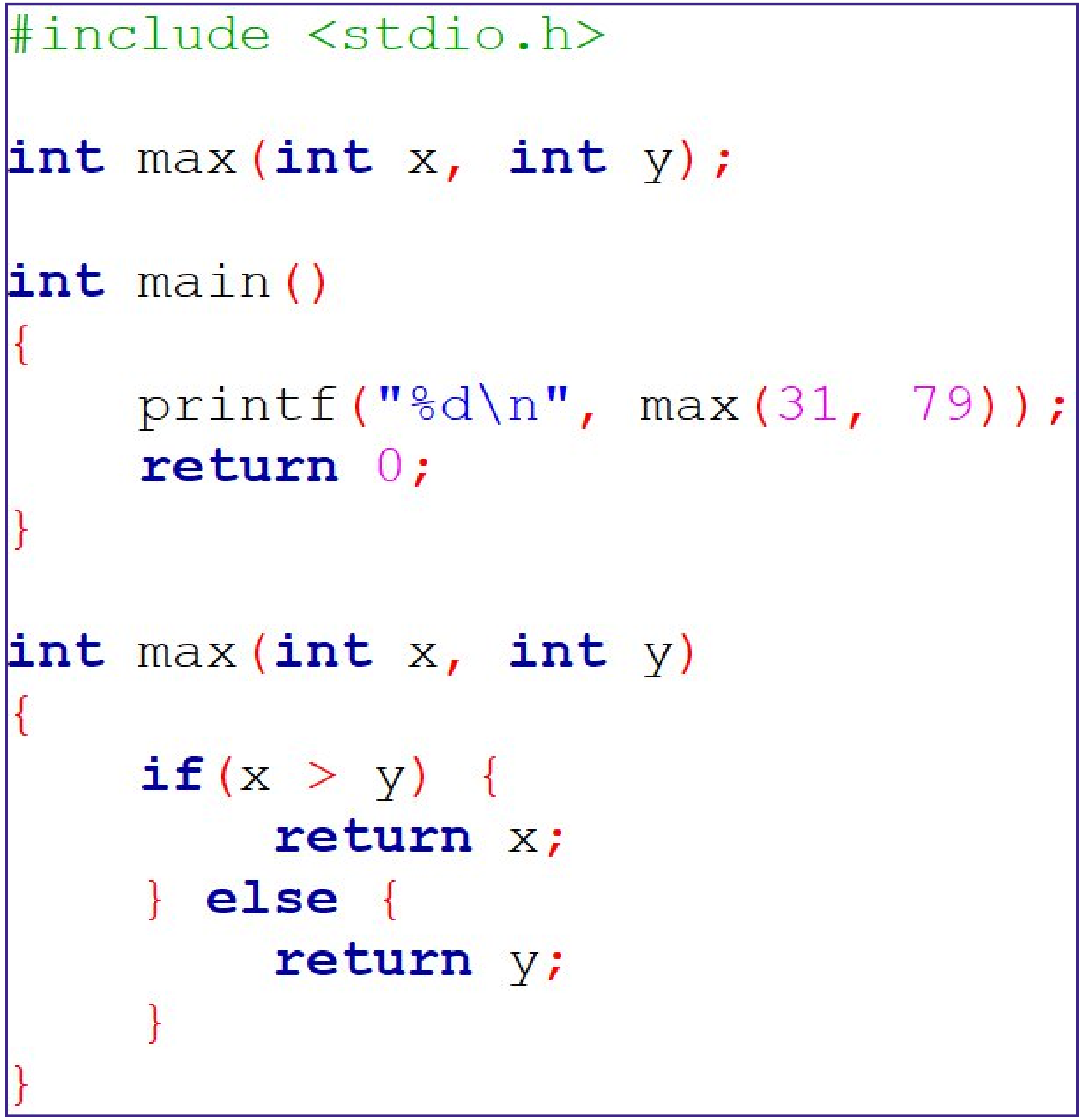
- It will be used to declare the functions, before implementing them: this will increase the code readability

1. Function Declaration

Function prototype



```
#include <stdio.h>

int max(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
}

int main()
{
    printf("%d\n", max(31, 79));
    return 0;
}
```



```
#include <stdio.h>

int max(int x, int y);

int main()
{
    printf("%d\n", max(31, 79));
    return 0;
}


int max(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
}
```

1. Function Declaration

Function prototype

- In such a simple case, the benefit is not obvious
- In a program with a lot of functions, it is better to do so
- The **main** appears first, which makes the code easier to read
- It also and especially allows to define functions **locally**

1. Function Declaration

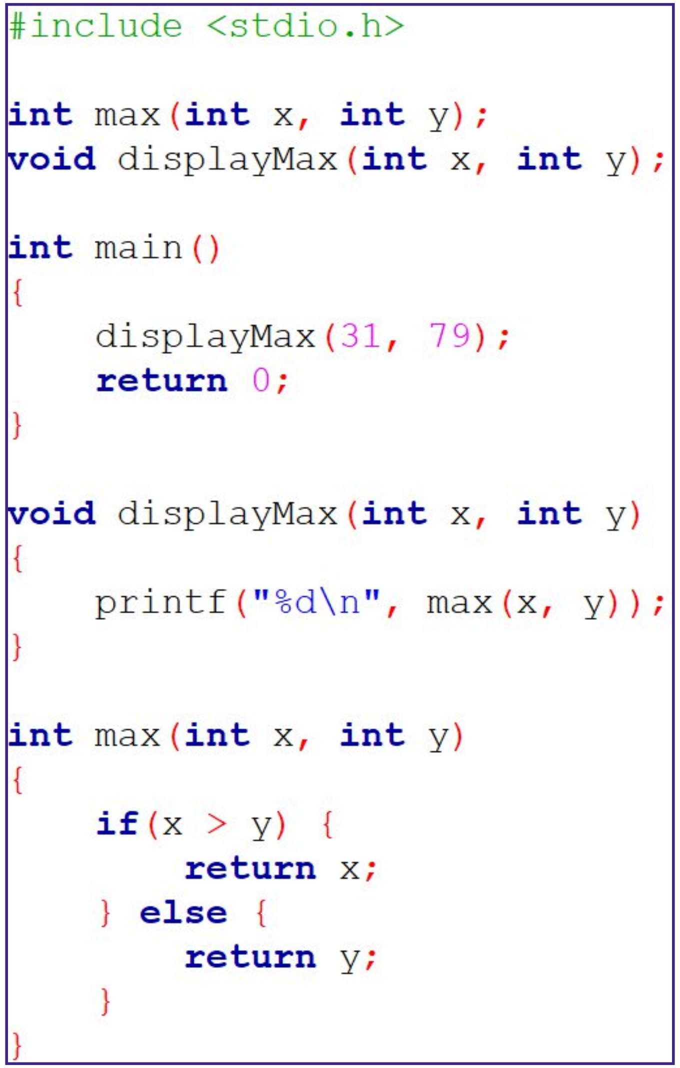
Global or local declaration

- For now, our function declarations are done before the **main**, in other words, in a **global** way

- This means that our functions were usable everywhere in the program, whether in **main** or in another function

1. Function Declaration

Global or local declaration



```c
#include <stdio.h>

int max(int x, int y);
void displayMax(int x, int y);

int main()
{
    displayMax(31, 79);
    return 0;
}

void displayMax(int x, int y)
{
    printf("%d\n", max(x, y));
}

int max(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
}
```

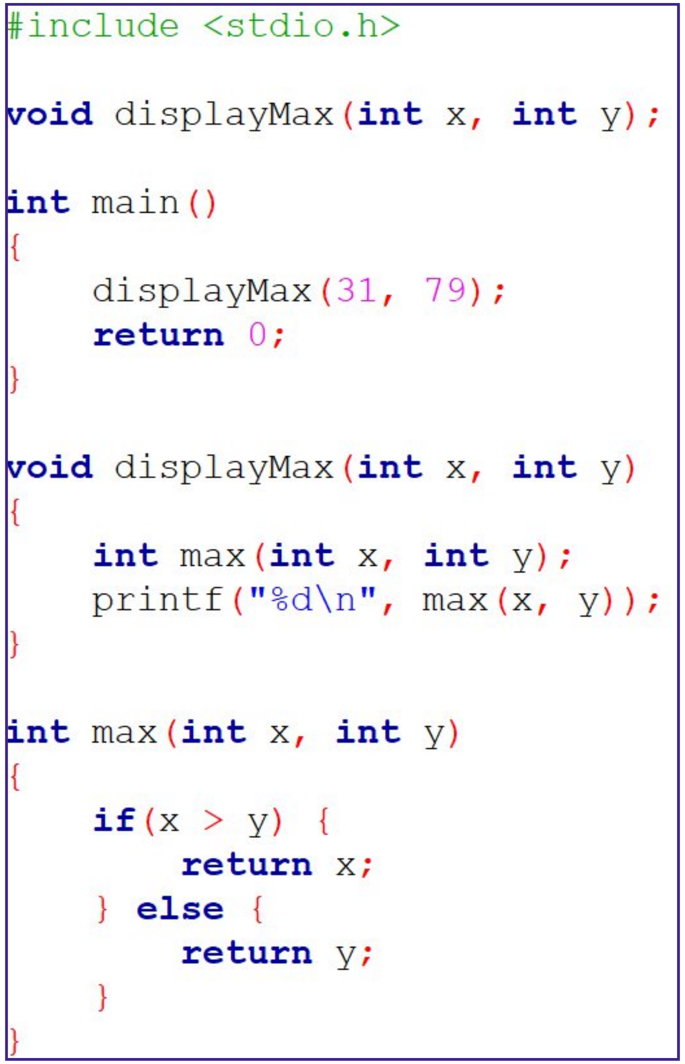
1. Function Declaration

Global or local declaration

- In this example, only the **displayMax** function uses the **max** function
- It therefore seems natural to define the **max** function only in the **displayMax** function
- We thus make a **local** declaration of **max**, which can then only be used in **displayMax**

1. Function Declaration

Global or local declaration

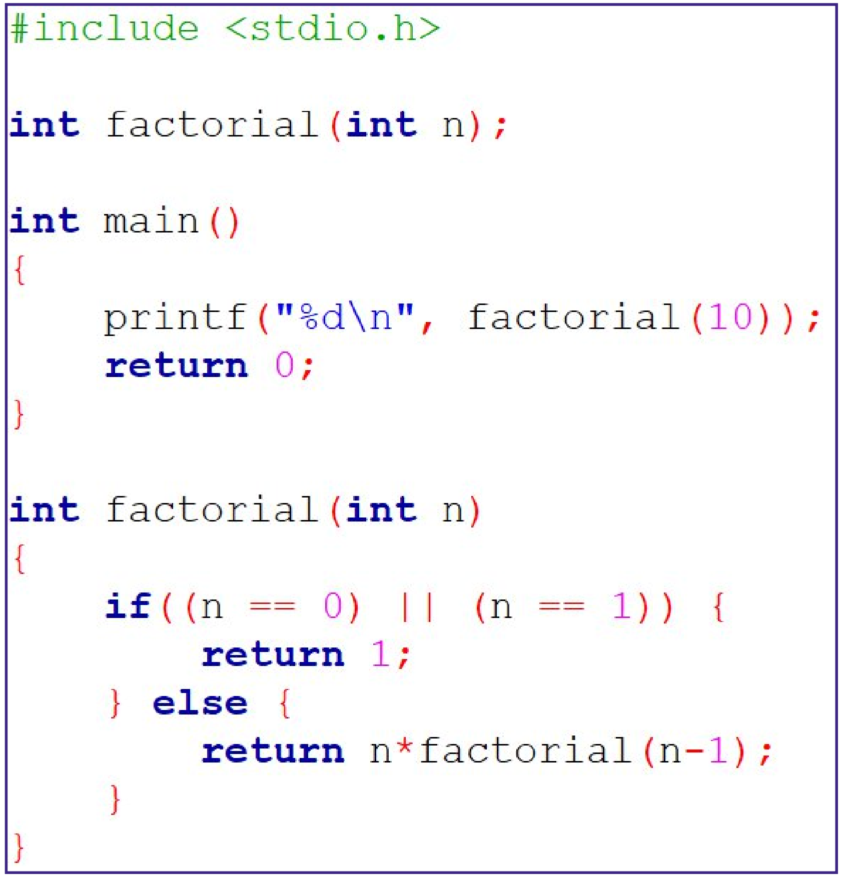


```c
#include <stdio.h>

void displayMax(int x, int y);

int main()
{
    displayMax(31, 79);
    return 0;
}

void displayMax(int x, int y)
{
    int max(int x, int y);
    printf("%d\n", max(x, y));
}

int max(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
}
```

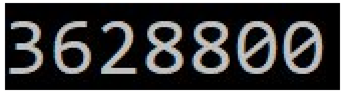
1. Function Declaration

Recursive functions

- Recursive functions can be implemented in C; but be careful to include a stop condition



```c
#include <stdio.h>

int factorial(int n);

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}

int factorial(int n)
{
    if((n == 0) || (n == 1)) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}
```



```
3628800
```

1. Function Declaration

Exercise

- You have a double and an integer
- Display the result of the first raised to the power of the second
- You will use a recursive function

1. Function Declaration

Questions

2. Splitting a Program into Modules

2. Splitting a Program into Modules

Principle

- For the moment, our projects have only one “**main.c**” file
- If the volume of a project becomes important, it will be divided into modules
- A module will simply be a coherent set of functions, usually around the same theme
- This will of course simplify the implementation of the project
- Another advantage is the possible reuse of a module from one project in another one

2. Splitting a Program into Modules

.h and .c files

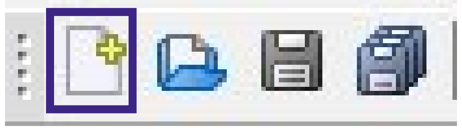
- We will of course keep our “**main.c**”, it will lead the project

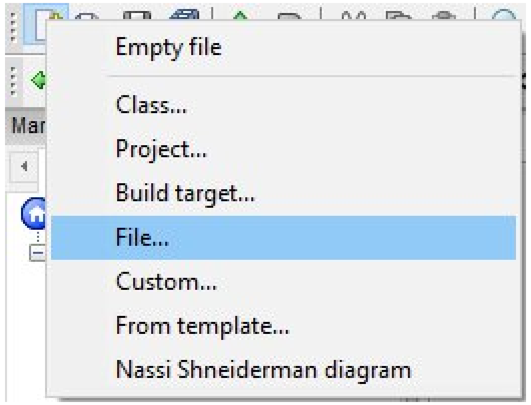
- Once we have conceptualized an intelligent breakdown, we will create two types of files for each module:

  – A **header**: “**.h**” file that will contain the function prototypes of the module
  – A **source file**: “**.c**” file that will contain their implementation

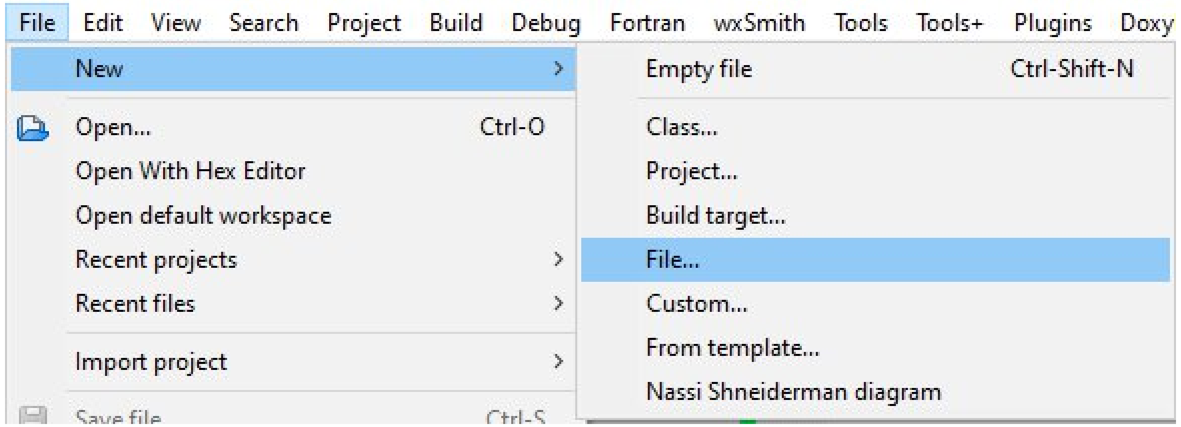
2. Splitting a Program into Modules

.h and .c files





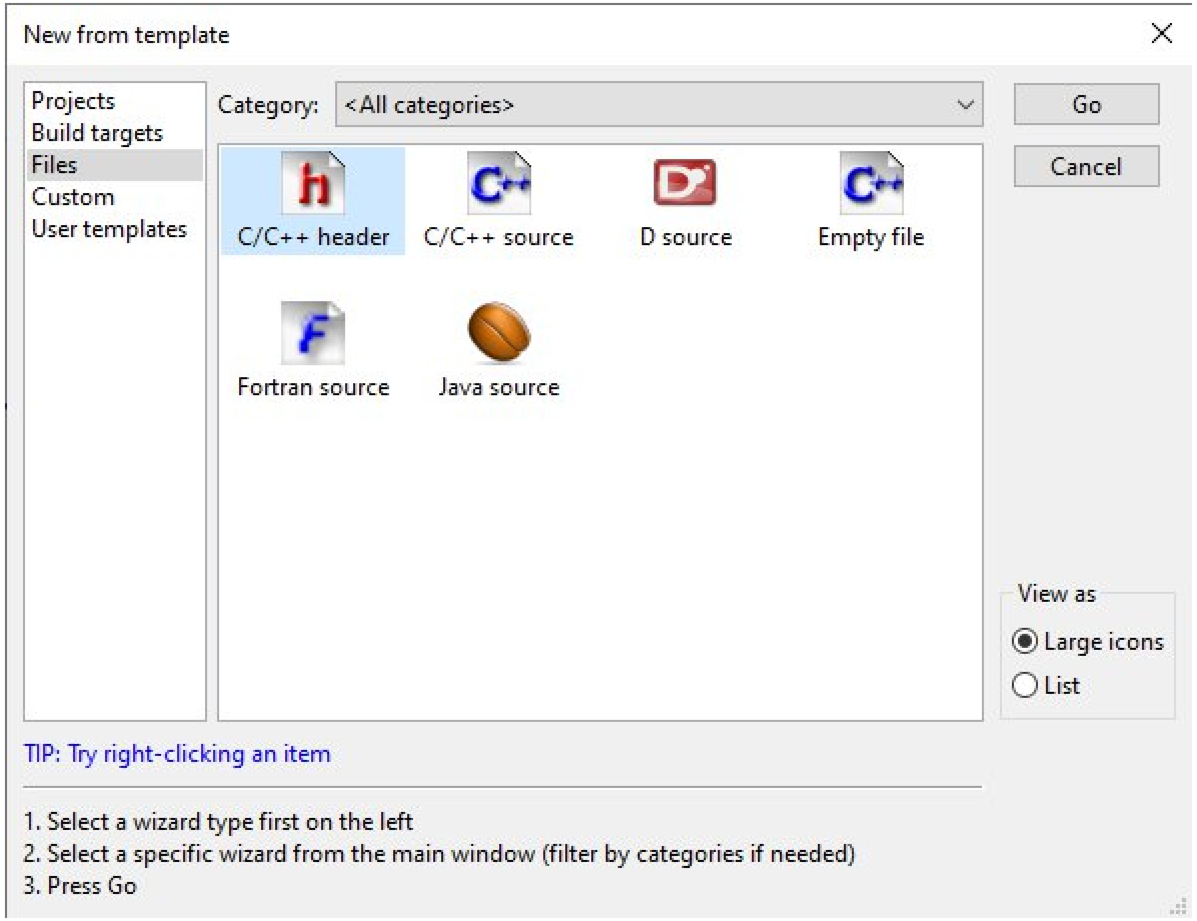
OR



2. Splitting a Program into Modules

.h and .c files

Choose the file type. First, add a “**.h**” file by selecting **C/C++ header**



New from template

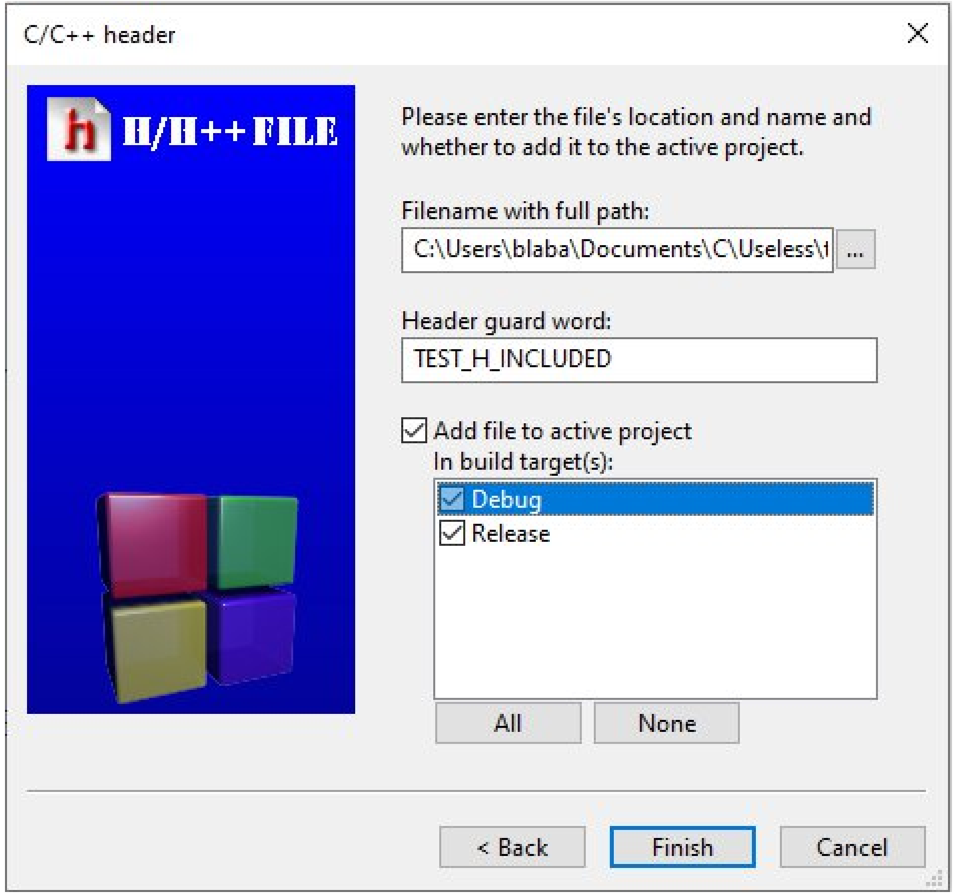
TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

2. Splitting a Program into Modules

.h and .c files

Enter the name and the directory in which you will save this new file, then select the build targets



Please enter the file's location and name and whether to add it to the active project.

Filename with full path: C:\Users\blaba\Documents\C\Useless\t

Header guard word: TEST_H_INCLUDED

☑ Add file to active project

In build target(s):

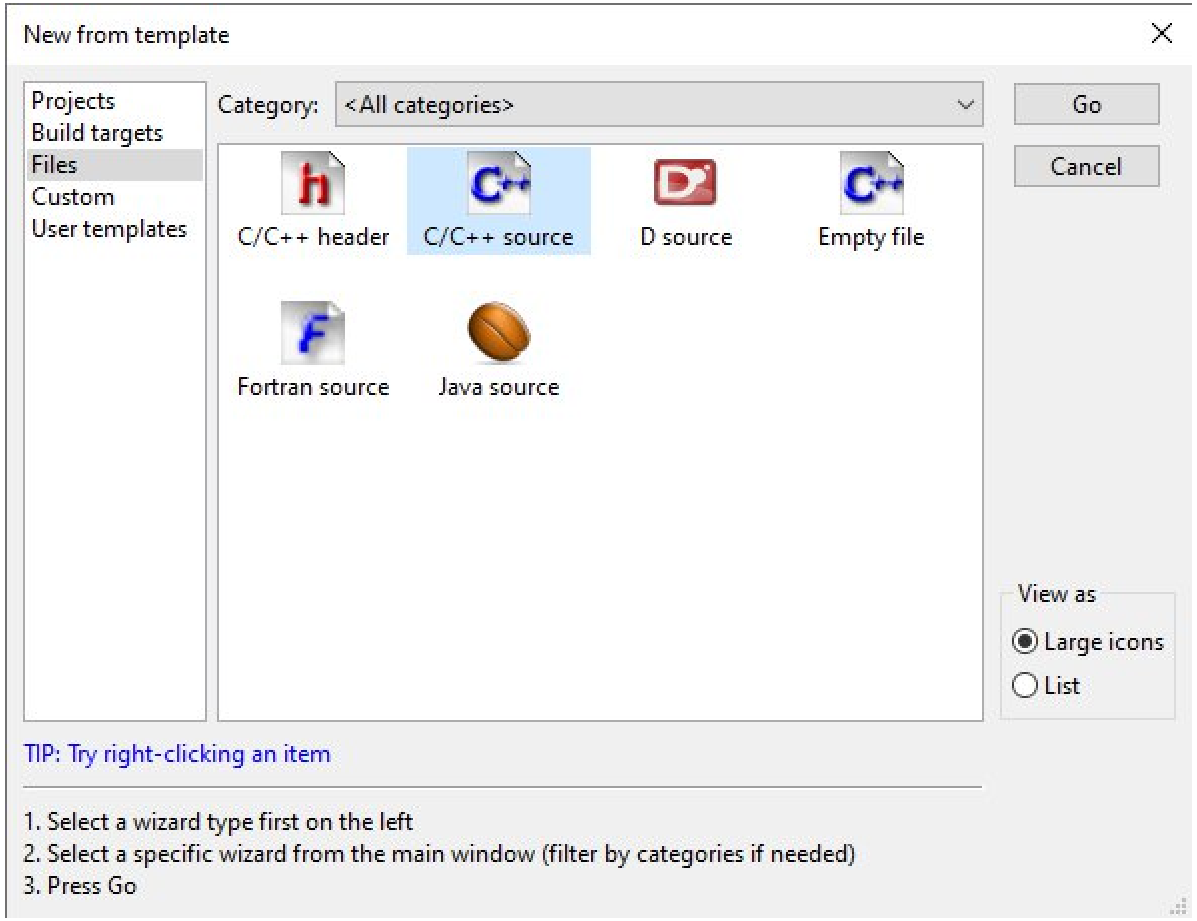
☑ Debug

☑ Release

2. Splitting a Program into Modules

.h and .c files

Second, add a “**.c**” file by selecting **C/C++ source**

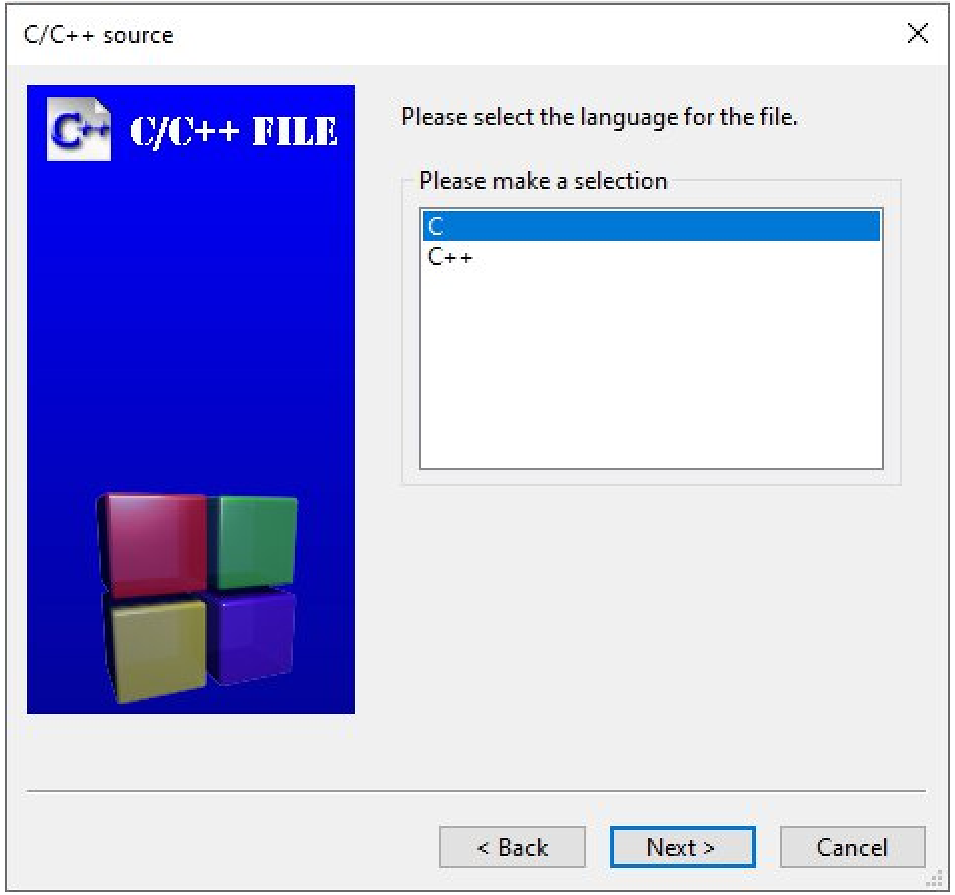


1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

2. Splitting a Program into Modules

.h and .c files

Same process, put an identical file name



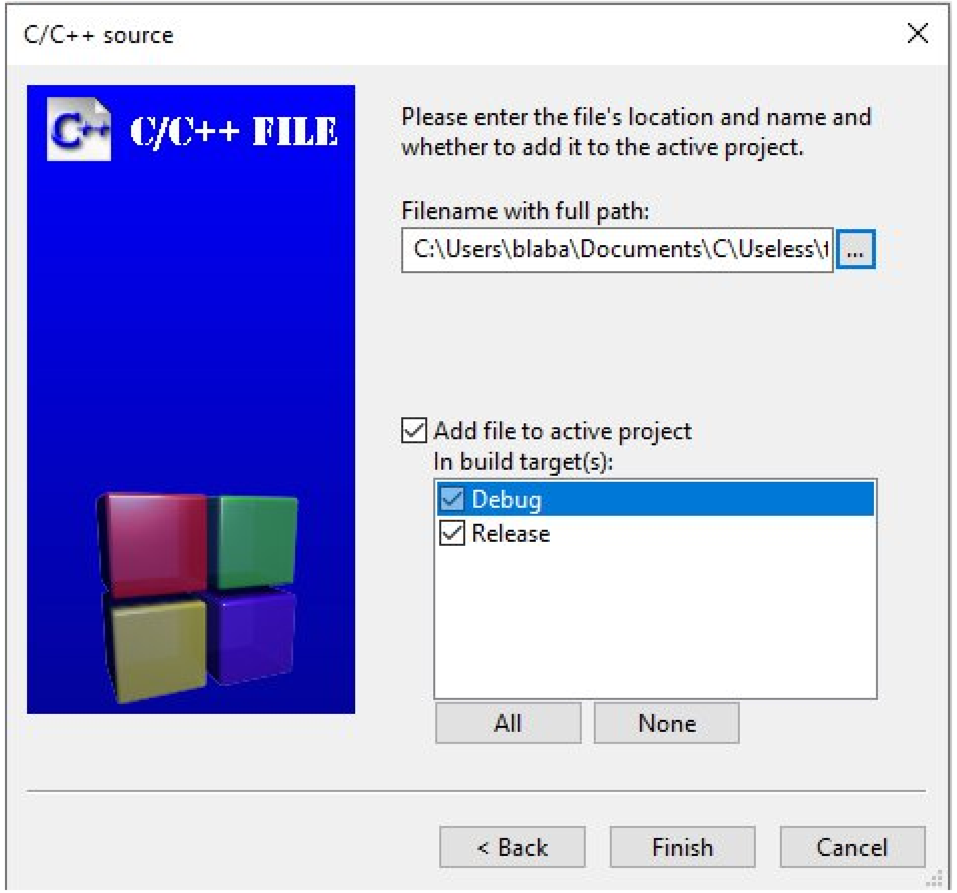
C/C++ source

Please select the language for the file.

Please make a selection

C

C++



C/C++ source

Please enter the file's location and name and whether to add it to the active project.

Filename with full path:

C:\Users\blaba\Documents\C\Useless\

☑ Add file to active project

In build target(s):

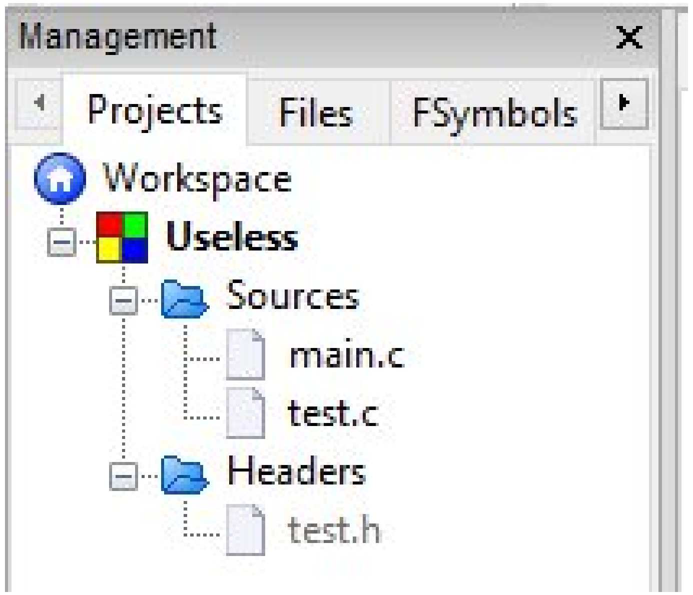
☑ Debug

☑ Release

2. Splitting a Program into Modules

.h and .c files

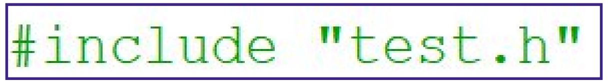
We now have two more files in our project and a new tree structure:



2. Splitting a Program into Modules

.h and .c files

- In order to use the functions that we are going to define in the **test** module, we need to include the “**test.h**” file into the **main**:



```
#include "test.h"
```

- This inclusion is not done with the same delimiters as the standard libraries

2. Splitting a Program into Modules

.h and .c files

- If our project contains several modules, it may happen that the same “**.h**” file is included in several other files

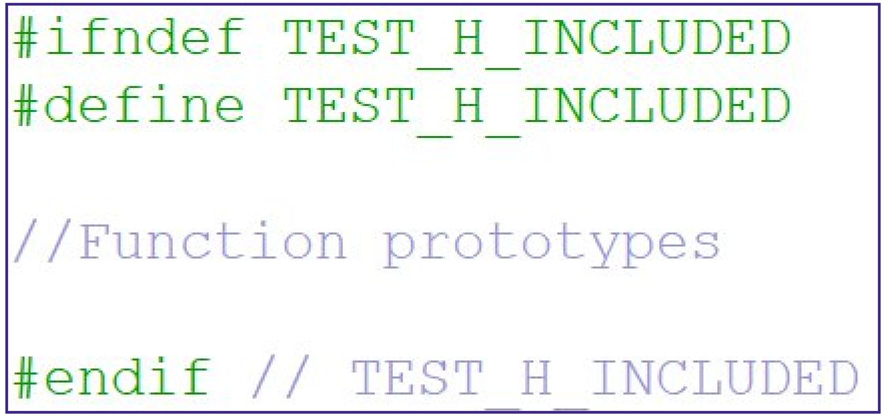
- This implies a risk of **multiple declarations** of functions in this file, and therefore of errors during compilation

- To avoid this phenomenon, it is thus necessary to set up a *protection* mechanism allowing to check if a “**.h**” file has already been included (thus if the declarations are already present in memory)

2. Splitting a Program into Modules

.h and .c files

“**.h**” content:



```
#ifndef TEST_H_INCLUDED
#define TEST_H_INCLUDED

//Function prototypes

#endif // TEST_H_INCLUDED
```

2. Splitting a Program into Modules

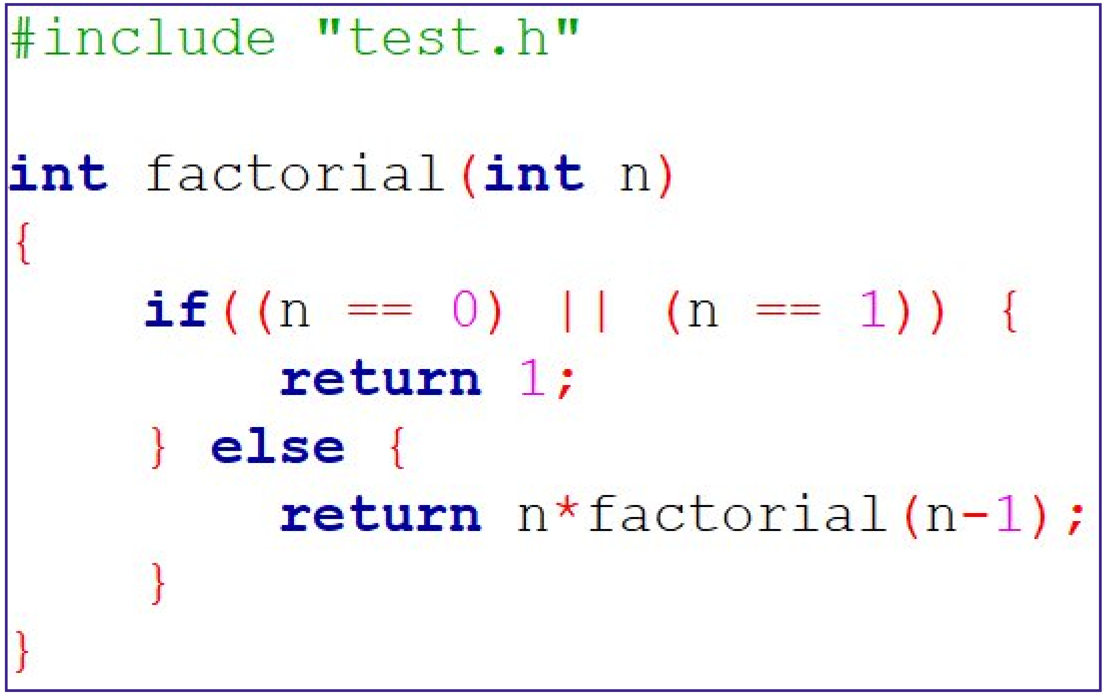
.h and .c files

- **#ifndef TEST_H_INCLUDED** checks if the constant **TEST_H_INCLUDED** has already been defined:
  - If it is not the case, we define it with **#define TEST_H_INCLUDED** then we load in memory the declarations of the functions contained in the file
  - If this is the case, it means that the file has already been included and that the function declarations are already in memory; we then go directly to **#endif** which indicates the end of the file

2. Splitting a Program into Modules

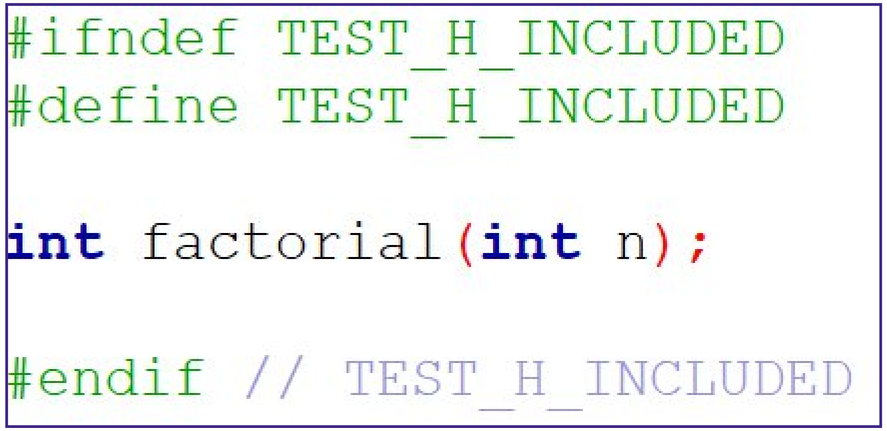
Example

test.c



```c
#include "test.h"

int factorial(int n)
{
    if((n == 0) || (n == 1)) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}
```

test.h

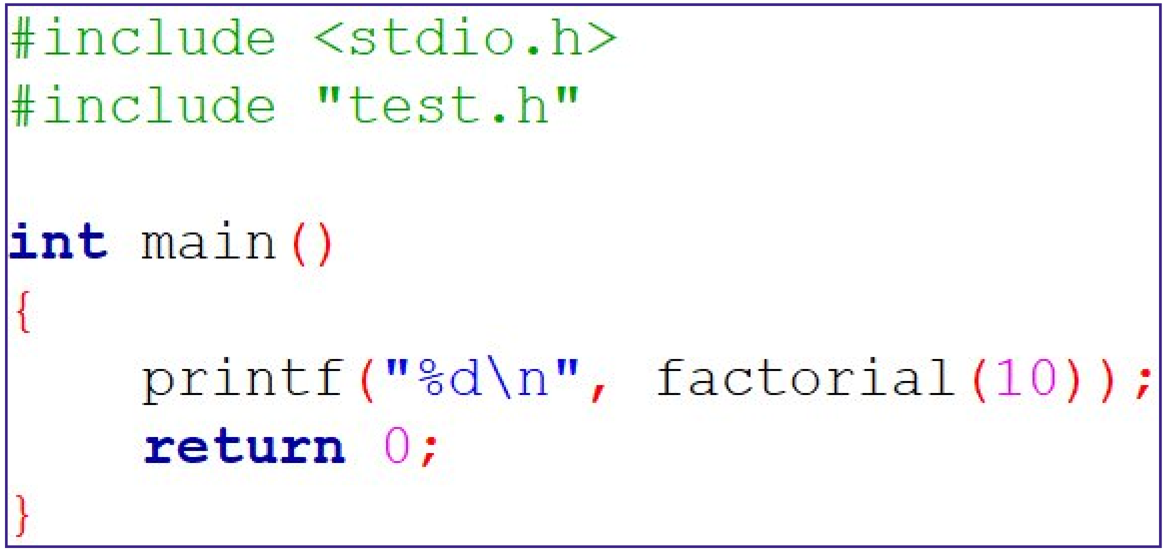


```c
#ifndef TEST_H_INCLUDED
#define TEST_H_INCLUDED

int factorial(int n);

#endif // TEST_H_INCLUDED
```

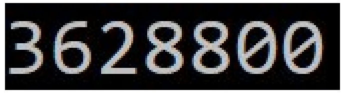
2. Splitting a Program into Modules

Example

main.c



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

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```



```
3628800
```

2. Splitting a Program into Modules

Questions

3. Function Parameters

3. Function Parameters

Passing arguments by value

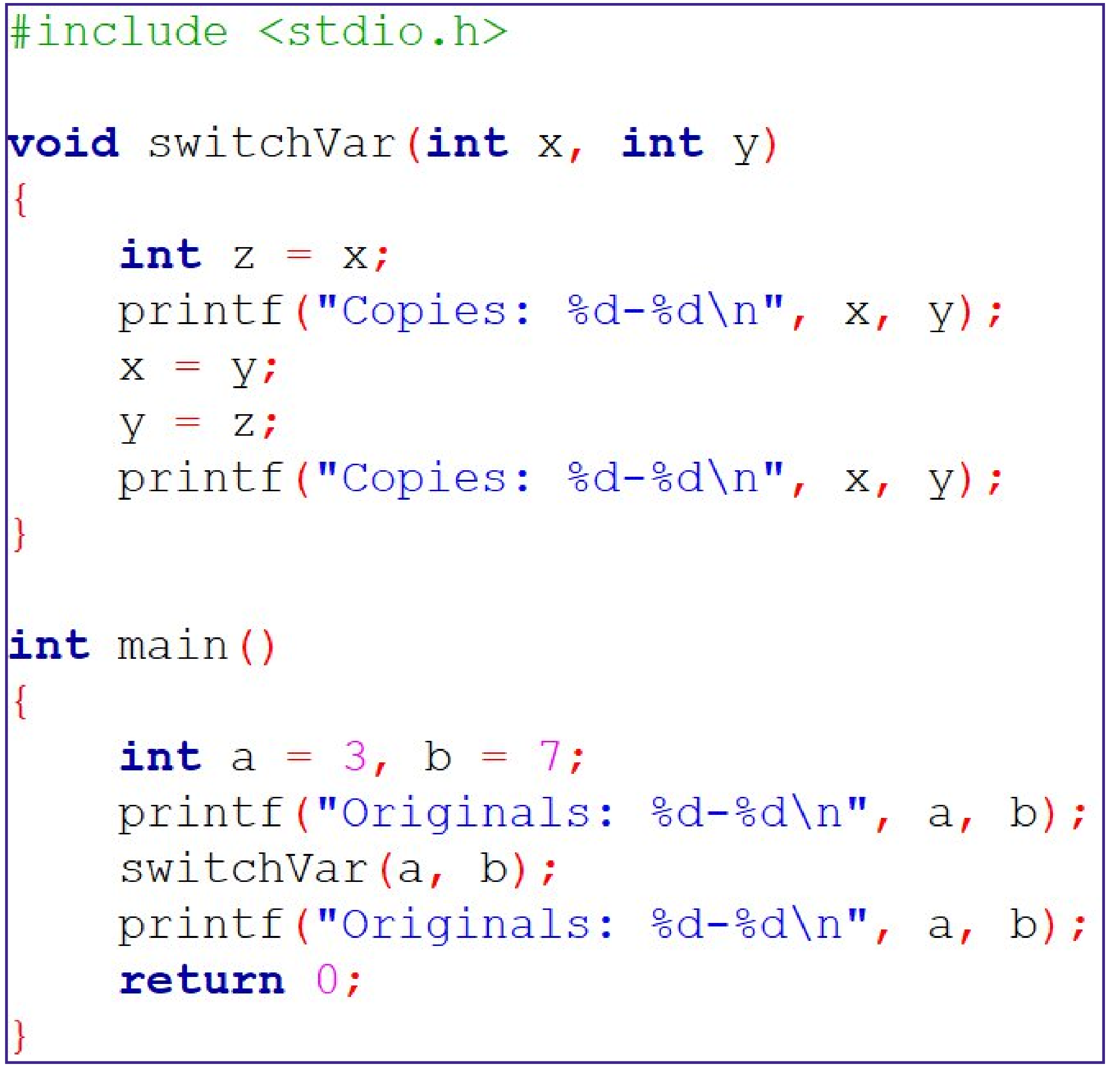
- When we send an argument to a function, a local variable is automatically created to store the value of this parameter

- The function then works on this **copy** and not on the **original**

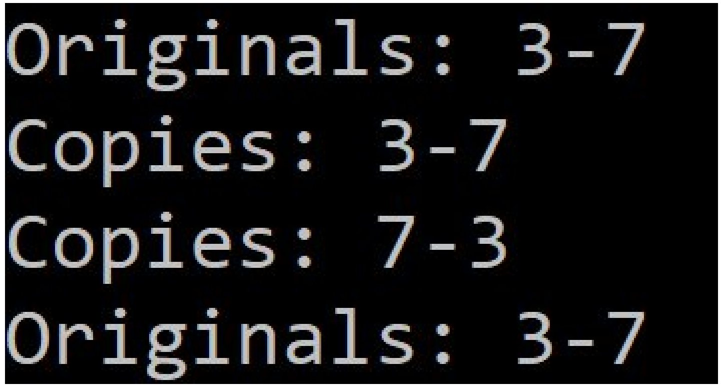
- If we pass a variable to a function, it will be able to modify the value of the local copy but not the original one

3. Function Parameters

Passing arguments by value



```c
#include <stdio.h>

void switchVar(int x, int y)
{
    int z = x;
    printf("Copies: %d-%d\n", x, y);
    x = y;
    y = z;
    printf("Copies: %d-%d\n", x, y);
}

int main()
{
    int a = 3, b = 7;
    printf("Originals: %d-%d\n", a, b);
    switchVar(a, b);
    printf("Originals: %d-%d\n", a, b);
    return 0;
}
```



```
Originals: 3-7
Copies: 3-7
Copies: 7-3
Originals: 3-7
```

3. Function Parameters

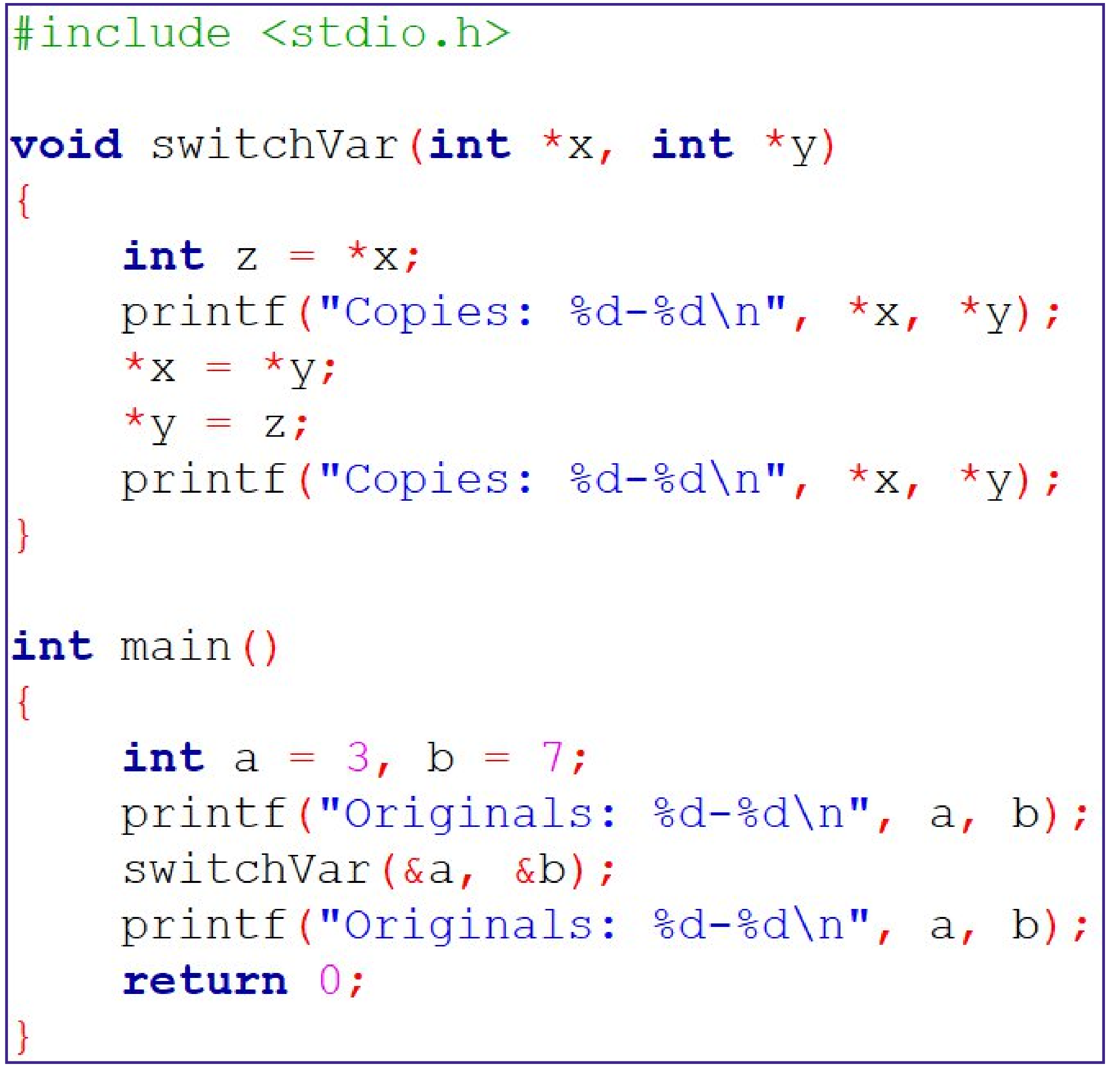
Passing arguments by pointer

- If we want a function to modify the value of a variable passed in argument, we must not transmit to the function the value of the variable but its address

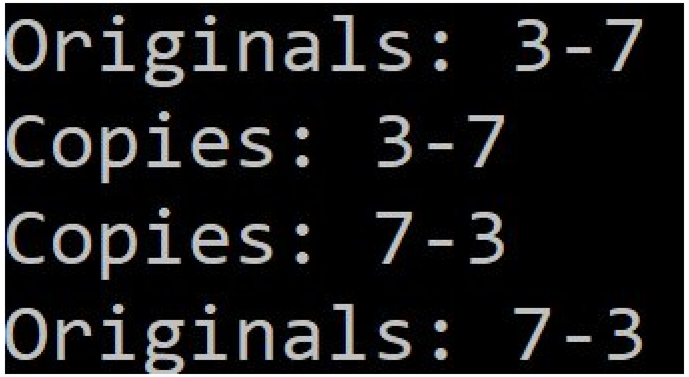
- The function will then work with a copy of the address of the variable, and can thus modify its value

3. Function Parameters

Passing arguments by pointer



```c
#include <stdio.h>

void switchVar(int *x, int *y)
{
    int z = *x;
    printf("Copies: %d-%d\n", *x, *y);
    *x = *y;
    *y = z;
    printf("Copies: %d-%d\n", *x, *y);
}

int main()
{
    int a = 3, b = 7;
    printf("Originals: %d-%d\n", a, b);
    switchVar(&a, &b);
    printf("Originals: %d-%d\n", a, b);
    return 0;
}
```



```
Originals: 3-7
Copies: 3-7
Copies: 7-3
Originals: 7-3
```

3. Function Parameters

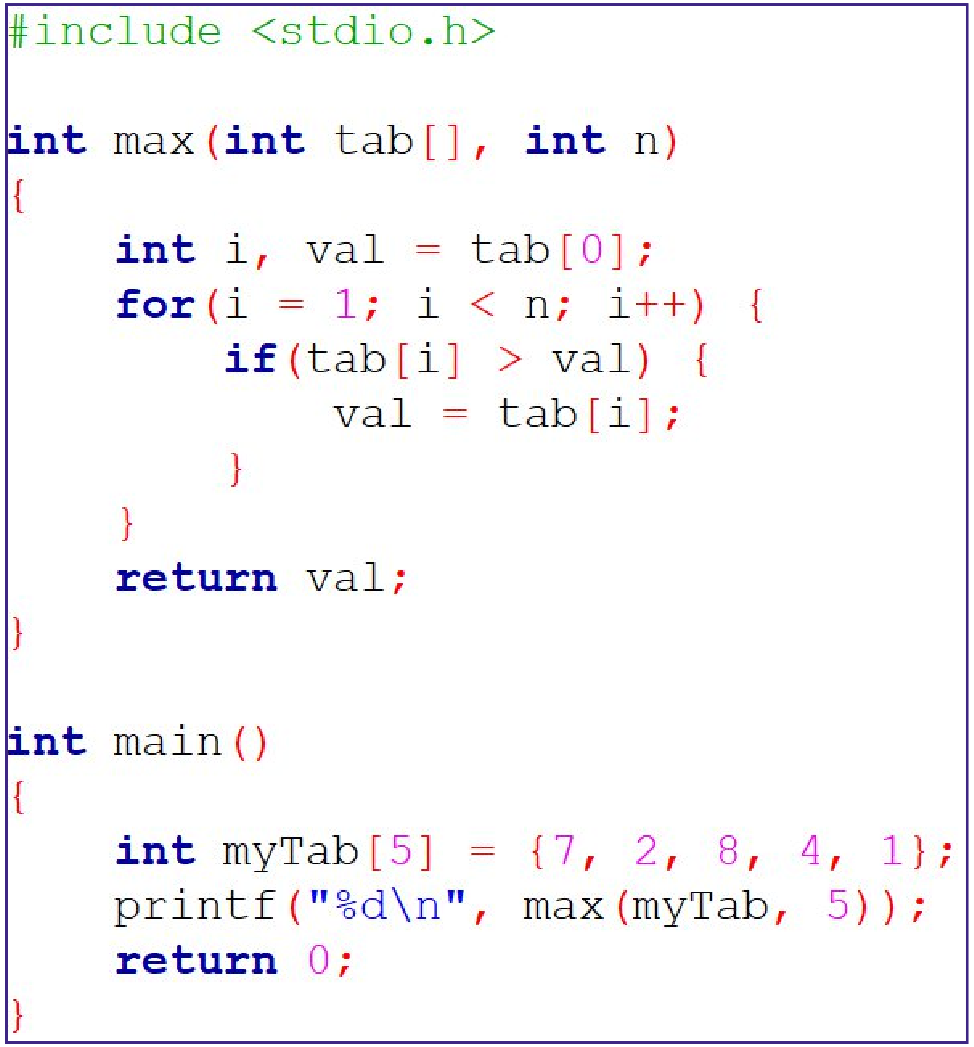
Arrays as parameters

- Using **type *tab;** or **type tab[];**, you must pass the size of the array as an argument

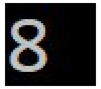
- It is therefore a question of passing arguments by pointer, and we can thus make modifications to this array

3. Function Parameters

Arrays as parameters



```c
#include <stdio.h>

int max(int tab[], int n)
{
    int i, val = tab[0];
    for(i = 1; i < n; i++) {
        if(tab[i] > val) {
            val = tab[i];
        }
    }
    return val;
}

int main()
{
    int myTab[5] = {7, 2, 8, 4, 1};
    printf("%d\n", max(myTab, 5));
    return 0;
}
```



```
8
```

3. Function Parameters

Structures as parameters

- No change compared to the passage of a variable of an elementary type
- The passage can be done by value or by pointer

3. Function Parameters

Exercise

- Write a function taking as parameters three real variables x, y, z
- Realize the circular permutation of these three variables: at the end of the function, x will contain the initial value of z, y will contain the initial value of x and z the initial value of y
- This function will not return any value

3. Function Parameters

Questions

C Developer

Functions

Thank you for your attention