Functions

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

University of Tlemcen
Department of Computer Science

https://sites.google.com/site/informatiquemessabihi/

Why Functions?

- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Why Functions?

- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Why Functions?

- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Why Functions?

- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Why Functions?

- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Why Functions?

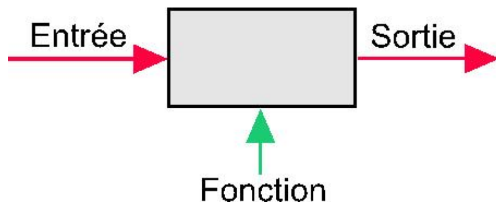
- A C language program begins with the main function.
- So far, we have stayed inside the main function. We have never exited it.
- It's not "wrong," but it's not what C programmers do in reality.
- Almost no program is written solely within the curly braces of the main function.
- So far, our programs were short, so it wasn't a big problem.
- But imagine larger programs with thousands of lines of code.

Solution: Function Concept

- Therefore, we need to learn to organize ourselves.
- We need to break down our programs into small pieces.
- Each "small piece of program" will be what we call a `function`.

Function

A function performs actions and returns a result. It is a piece of code that serves to do something specific.

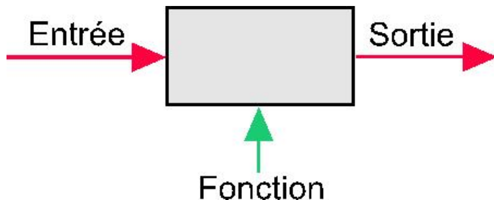


Solution: Function Concept

- Therefore, we need to learn to organize ourselves.
- We need to break down our programs into small pieces.
- Each "small piece of program" will be what we call a `function`.

Function

A function performs actions and returns a result. It is a piece of code that serves to do something specific.



Advantages of Functions

Functions are independent modules (groups of instructions) designated by a name. They have several advantages:

1. They allow **"factorizing"** programs, i.e., sharing common parts.
2. They enable **structuring** and **improving readability** of programs.
3. They **simplify code maintenance** (just needs to be modified once).
4. They can potentially be **reused** in other programs.

Advantages of Functions

Functions are independent modules (groups of instructions) designated by a name. They have several advantages:

1. They allow **"factorizing"** programs, i.e., sharing common parts.
2. They enable **structuring** and **improving readability** of programs.
3. They **simplify code maintenance** (just needs to be modified once).
4. They can potentially be **reused** in other programs.

Advantages of Functions

Functions are independent modules (groups of instructions) designated by a name. They have several advantages:

1. They allow **"factorizing"** programs, i.e., sharing common parts.
2. They enable **structuring** and **improving readability** of programs.
3. They **simplify code maintenance** (just needs to be modified once).
4. They can potentially be **reused** in other programs.

Advantages of Functions

Functions are independent modules (groups of instructions) designated by a name. They have several advantages:

1. They allow **"factorizing"** programs, i.e., sharing common parts.
2. They enable **structuring** and **improving readability** of programs.
3. They **simplify code maintenance** (just needs to be modified once).
4. They can potentially be **reused** in other programs.

Advantages of Functions

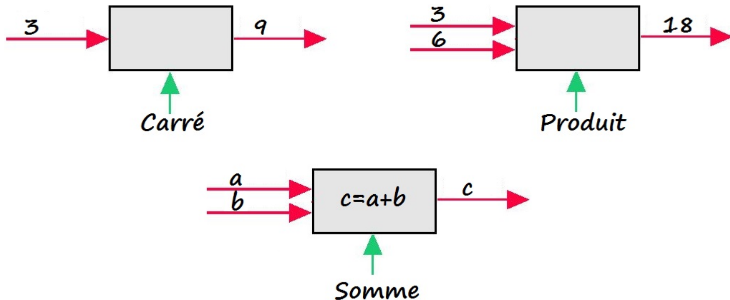
Functions are independent modules (groups of instructions) designated by a name. They have several advantages:

1. They allow **"factorizing"** programs, i.e., sharing common parts.
2. They enable **structuring** and **improving readability** of programs.
3. They **simplify code maintenance** (just needs to be modified once).
4. They can potentially be **reused** in other programs.

Principle

A function is defined by three elements:

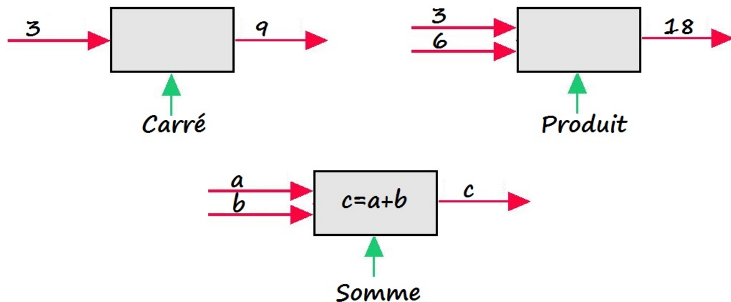
1. `Input:` We "input" information into the function (providing it with data to work on).
2. `Calculations:` With the input information, the function performs its work.
3. `Output:` Once it has finished its calculations, the function returns a result. This is called the output or return.



Principle

A function is defined by three elements:

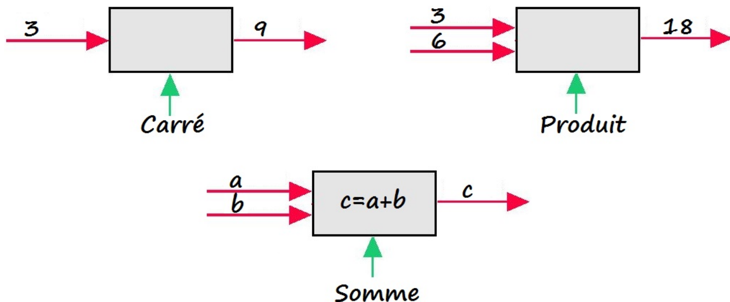
1. `Input`: We "input" information into the function (providing it with data to work on).
2. `Calculations`: With the input information, the function performs its work.
3. `Output`: Once it has finished its calculations, the function returns a result. This is called the output or return.



Principle

A function is defined by three elements:

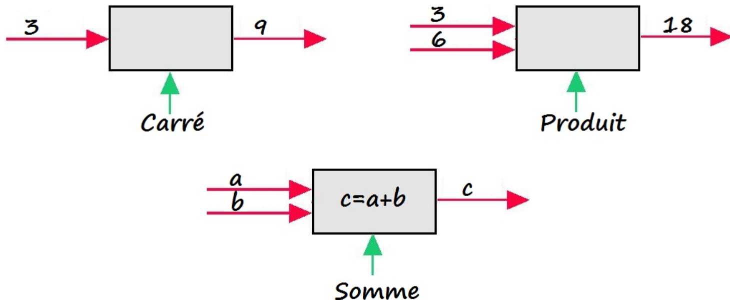
1. `Input:` We "input" information into the function (providing it with data to work on).
2. `Calculations:` With the input information, the function performs its work.
3. `Output:` Once it has finished its calculations, the function returns a result. This is called the output or return.



Principle

A function is defined by three elements:

1. `Input:` We "input" information into the function (providing it with data to work on).
2. `Calculations:` With the input information, the function performs its work.
3. `Output:` Once it has finished its calculations, the function returns a result. This is called the output or return.



Declaring a Function

Syntax:

```
<Return_Type> <Function_Name> (<Parameters>)
   {
    <Function_Body>
   }
```

- `Return type:` (corresponds to the output) it is the type of the function. This type depends on the result that the function returns (**int**, **double**, **void**,...)
- `Function Name:` this is the name of your function. It must follow the same rules as variables.
- `Parameters:` (corresponds to the input) in parentheses, we send parameters to the function.

Declaring a Function

Syntax:

```
<Return_Type> <Function_Name> (<Parameters>)
   {
    <Function_Body>
   }
```

- `Return type:` (corresponds to the output) it is the type of the function. This type depends on the result that the function returns (**int**, **double**, **void**,...)
- `Function Name:` this is the name of your function. It must follow the same rules as variables.
- `Parameters:` (corresponds to the input) in parentheses, we send parameters to the function.

Declaring a Function

Syntax:

```
<Return_Type> <Function_Name> (<Parameters>)
   {
    <Function_Body>
   }
```

- `Return type:` (corresponds to the output) it is the type of the function. This type depends on the result that the function returns (**int**, **double**, **void**,...)
- `Function Name:` this is the name of your function. It must follow the same rules as variables.
- `Parameters:` (corresponds to the input) in parentheses, we send parameters to the function.

Declaring a Function

Syntax:

```
<Return_Type> <Function_Name> (<Parameters>)
   {
    <Function_Body>
   }
```

- `Return type:` (corresponds to the output) it is the type of the function. This type depends on the result that the function returns (**int**, **double**, **void**,...)
- `Function Name:` this is the name of your function. It must follow the same rules as variables.
- `Parameters:` (corresponds to the input) in parentheses, we send parameters to the function.

Void Return Type

- It may be necessary to code a function that does not return any result.
- This is a common case in C. This type of function is called a procedure.
- To write a procedure, you need to indicate to the function that it should not return anything.
- For this, there is a special "return type": **void**. This type means "empty" and is used to indicate that the function has no result.

Example

```
void displayMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Black Coffee \n");
        printf("2. Latte \n");
        printf("3. Hot Chocolate\n");
        printf("4. Mint Tea \n");
    }
```

Void Return Type

- It may be necessary to code a function that does not return any result.
- This is a common case in C. This type of function is called a procedure.
- To write a procedure, you need to indicate to the function that it should not return anything.
- For this, there is a special "return type": **void**. This type means "empty" and is used to indicate that the function has no result.

Example

```
void displayMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Black Coffee \n");
        printf("2. Latte \n");
        printf("3. Hot Chocolate\n");
        printf("4. Mint Tea \n");
    }
```

Void Return Type

- It may be necessary to code a function that does not return any result.
- This is a common case in C. This type of function is called a procedure.
- To write a procedure, you need to indicate to the function that it should not return anything.
- For this, there is a special "return type": **void**. This type means "empty" and is used to indicate that the function has no result.

Example

```
void displayMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Black Coffee \n");
        printf("2. Latte \n");
        printf("3. Hot Chocolate\n");
        printf("4. Mint Tea \n");
    }
```

Void Return Type

- It may be necessary to code a function that does not return any result.
- This is a common case in C. This type of function is called a procedure.
- To write a procedure, you need to indicate to the function that it should not return anything.
- For this, there is a special "return type": **void**. This type means "empty" and is used to indicate that the function has no result.

Example

```
void displayMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Black Coffee \n");
        printf("2. Latte \n");
        printf("3. Hot Chocolate\n");
        printf("4. Mint Tea \n");
    }
```

Void Return Type

- It may be necessary to code a function that does not return any result.
- This is a common case in C. This type of function is called a procedure.
- To write a procedure, you need to indicate to the function that it should not return anything.
- For this, there is a special "return type": **void**. This type means "empty" and is used to indicate that the function has no result.

Example

```
void displayMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Black Coffee \n");
        printf("2. Latte \n");
        printf("3. Hot Chocolate\n");
        printf("4. Mint Tea \n");
    }
```

Function Parameters

- A parameter serves to provide information to the function during its execution.
- If the function requires multiple parameters, separate them with commas.

Example:

```
int sum(int a, int b)
{
    return a + b;
}
// Functions without parameters
void greet()
{
    printf("Hello");
}
```

- Parameters must have different names.
- It is also possible to have no arguments in a function. In this case, write **()** or **(void)**.

Function Parameters

- A parameter serves to provide information to the function during its execution.
- If the function requires multiple parameters, separate them with commas.

Example:

```
int sum(int a, int b)
{
    return a + b;
}
// Functions without parameters
void greet()
{
    printf("Hello");
}
```

- Parameters must have different names.
- It is also possible to have no arguments in a function. In this case, write **()** or **(void)**.

Function Parameters

- A parameter serves to provide information to the function during its execution.
- If the function requires multiple parameters, separate them with commas.

Example:

```
int sum(int a, int b)
{
    return a + b;
}
// Functions without parameters
void greet()
{
    printf("Hello");
}
```

- Parameters must have different names.
- It is also possible to have no arguments in a function. In this case, write **()** or **(void)**.

Function Parameters

- A parameter serves to provide information to the function during its execution.
- If the function requires multiple parameters, separate them with commas.

Example:

```
int sum(int a, int b)
{
    return a + b;
}
// Functions without parameters
void greet()
{
    printf("Hello");
}
```

- Parameters must have different names.
- It is also possible to have no arguments in a function. In this case, write **()** or **(void)**.

Function Parameters

- A parameter serves to provide information to the function during its execution.
- If the function requires multiple parameters, separate them with commas.

Example:

```
int sum(int a, int b)
{
    return a + b;
}
// Functions without parameters
void greet()
{
    printf("Hello");
}
```

- Parameters must have different names.
- It is also possible to have no arguments in a function. In this case, write **()** or **(void)**.

Return Statement

- The `return` statement specifies the result that the function should return (send back).
- Any expression can be mentioned after `return`.

Example:

```
float polynomial(float x, int b, int c)
 {
 float result;
 result = x * x + b * x + c;
 return result;

 // is equivalent to

 float polynomial(float x, int b, int c)
 {
 return (x * x + b * x + c);
 }
```

Return Statement

- The `return` statement can appear multiple times in a function.

Example:

```
double absoluteProduct(double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s);
 }
```

- The type of the expression in `return` must be the same as declared in the function header. Otherwise, the compiler will automatically insert conversion instructions.

Return Statement

- The `return` statement can appear multiple times in a function.

Example:

```
double absoluteProduct(double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s);
 }
```

- The type of the expression in `return` must be the same as declared in the function header. Otherwise, the compiler will automatically insert conversion instructions.

Return Statement

- The `return` statement can appear multiple times in a function.

Example:

```
double absoluteProduct(double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s);
 }
```

- The type of the expression in `return` must be the same as declared in the function header. Otherwise, the compiler will automatically insert conversion instructions.

Usage of a Function

Simply type the name of the function followed by the parameters in parentheses.

Example:

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

int triple(int number) // 6
{
    return 3 * number; // 7
}
int main() // 1
{
  int enteredNumber = 0, tripledNumber = 0; // 2
  printf("Enter a number... "); // 3
  scanf("%d", &enteredNumber); // 4

  tripledNumber = triple(enteredNumber); // 5

  printf("The triple of this number is %d\n", tripledNumber)
      ;//8
  return 0; // 9
}
```

Function Call

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

int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
    int nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %d\n", nombreTriple);

    return 0;
}
```

1

2

Not Required to Store the Result of a Function

Example:

```
int triple(int number)
{
    return 3 * number;
}
int main()
{
        ...
        printf("The triple is %d\n", triple(inputNumber));
        ...
}
```

The `main` function calls the `printf` function, which in turn calls the `triple` function. It's a nesting of functions.

Formal Parameters Vs. Actual Parameters

```
int triple(int number)
{
    return 3 * number;
}

int main()
{
        ...
        printf("The triple is %d\n", triple(inputNumber));
        ...
}
```

1. The names of the arguments in the function header are called "formal parameters." Their role is to describe what the function should do within its body.
2. The arguments provided during the use (the call) of the function are called "actual parameters." Any expression can be used as an actual parameter.

Formal Parameters Vs. Actual Parameters

```
int triple(int number)
{
    return 3 * number;
}

int main()
{
        ...
        printf("The triple is %d\n", triple(inputNumber));
        ...
}
```

1. The names of the arguments in the function header are called "formal parameters." Their role is to describe what the function should do within its body.
2. The arguments provided during the use (the call) of the function are called "actual parameters." Any expression can be used as an actual parameter.

Passing Parameters by Value

Example:

```
#include <stdio.h>

void function(int number)
{
    ++number;
    printf("Variable 'number' in the function: %d\n", number
        );
}
int main(void)
{
    int number = 5;
    function(number);
    printf("Variable 'number' in main: %d\n", number);
    return 0;
}
```

Passing Parameters by Value

Example:

```
#include <stdio.h>

void function(int number)
{
    ++number;
    printf("Variable 'number' in the function: %d\n", number
        );
}
int main(void)
{
    int number = 5;
    function(number);
    printf("Variable 'number' in main: %d\n", number);
    return 0;
}
```

Variable 'number' in the function: 6

Passing Parameters by Value

Example:

```
#include <stdio.h>

void function(int number)
{
    ++number;
    printf("Variable 'number' in the function: %d\n", number
        );
}
int main(void)
{
    int number = 5;
    function(number);
    printf("Variable 'number' in main: %d\n", number);
    return 0;
}
```

Variable 'number' in the function: 6
Variable 'number' in main: 5

Scope of Variables

```
#include <stdio.h>
int i    =        4;
int f1(int a){
  i      =        i*f2(i-1);
  return i;
}
int f2(int i){
  i      =        i*f3(i-1);
  return i;
}
int f3(int a){
  int i =         4;
  i      =        i        *        (i-1);
  return i;
}
void main(){
  int i =         0;
  while(i<3){
     printf("Enter an integer value\n");
     scanf("%d",          &i); }
  i      =        f1(i);
  printf("The result of the program is: %d\n", i);
}
```

Function Prototypes

- Defining a function after `main` will cause undefined behavior.
- Compilation could work or crash.
- In principle, the order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In the prototype parameters, only the types are really necessary; identifiers are optional.

Function Prototypes

- Defining a function after `main` will cause undefined behavior.
- Compilation could work or crash.
- In principle, the order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In the prototype parameters, only the types are really necessary; identifiers are optional.

Function Prototypes

- Defining a function after `main` will cause undefined behavior.

- Compilation could work or crash.

- In principle, the order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.

- In the prototype parameters, only the types are really necessary; identifiers are optional.

Function Prototypes

- Defining a function after `main` will cause undefined behavior.
- Compilation could work or crash.
- In principle, the order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In the prototype parameters, only the types are really necessary; identifiers are optional.

Function Prototypes

Example:

```
#include <stdio.h>
int square(int number);

int main(void)
{
    int number, squaredNumber;
    puts("Enter a number:");
    scanf("%d", &number);
    squaredNumber = square(number);
    printf("Here is the square of %d: %d\n", number,
        squaredNumber);
    return 0;
}

int square(int number)
{
    number *= number;
    return number;
}
```

Remarks

- The default type is `int`; in other words, if the type of a function is not explicitly declared, it is automatically `int`.
- It is forbidden to define functions inside another function (unlike Pascal).
- The order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In function prototype parameters, only the types are necessary; identifiers are optional.

Remarks

- The default type is `int`; in other words, if the type of a function is not explicitly declared, it is automatically `int`.
- It is forbidden to define functions inside another function (unlike Pascal).
- The order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In function prototype parameters, only the types are necessary; identifiers are optional.

Remarks

- The default type is `int`; in other words, if the type of a function is not explicitly declared, it is automatically `int`.

- It is forbidden to define functions inside another function (unlike Pascal).

- The order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.

- In function prototype parameters, only the types are necessary; identifiers are optional.

Remarks

- The default type is `int`; in other words, if the type of a function is not explicitly declared, it is automatically `int`.
- It is forbidden to define functions inside another function (unlike Pascal).
- The order of definitions in the program text does not play a role, but each function must be declared (prototyped) or defined before being called.
- In function prototype parameters, only the types are necessary; identifiers are optional.