FUNCTIONS, typedef, POINTERS to FUNCTIONS, STACKS and RECURSION.

Mathematics is the Science of Analogies.
Sir Michael Atiyah ("Fields Medal", 1966)

5.1 Functions.

In a program, it is almost always necessary to perform a specific task that is repeated many times or a task involving a complex algorithm that, for security reasons or better program design, needs to be encapsulated. It also often happens, that a task that is performed in one program is useful for other different programs.

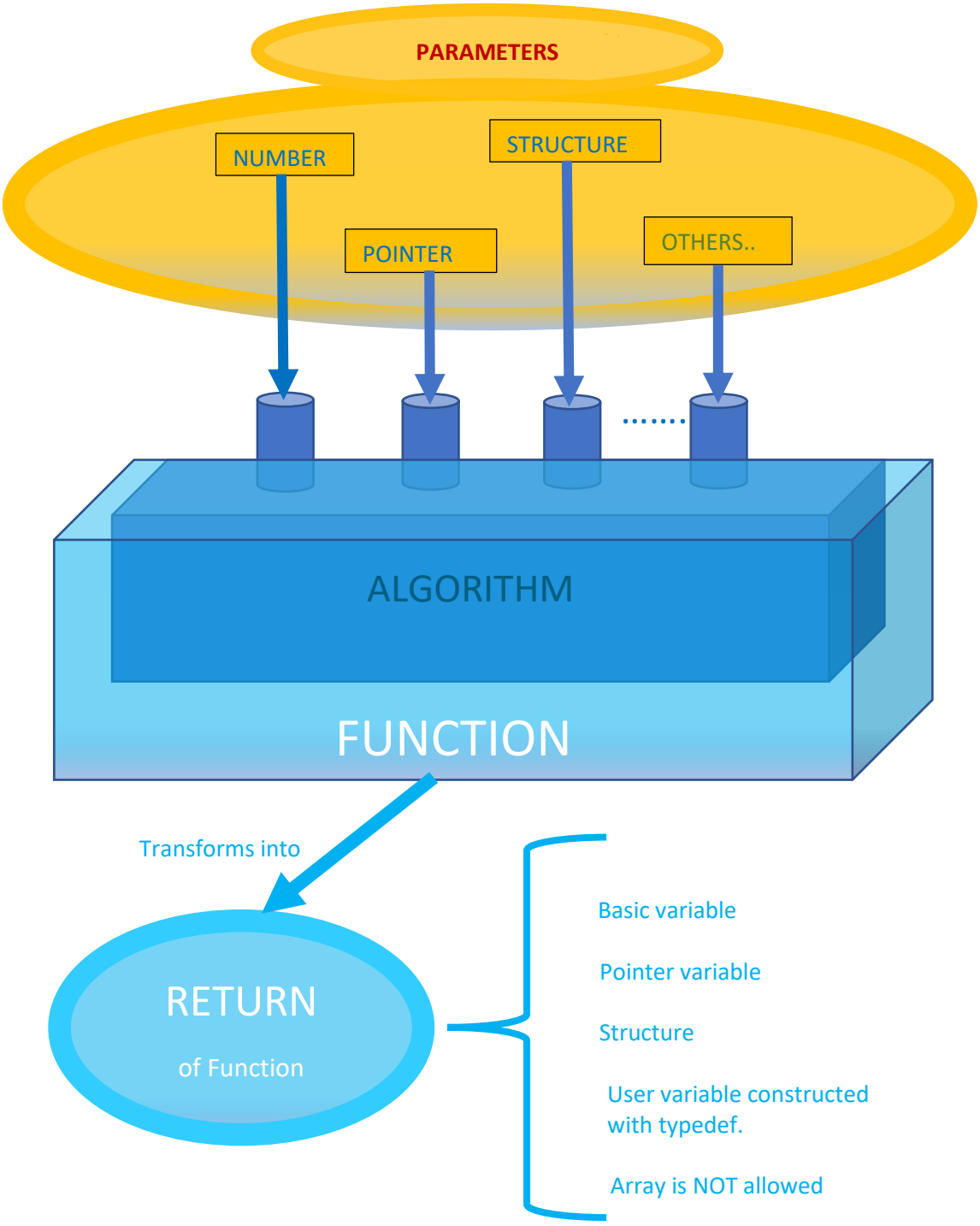
The C language provides what it calls **Functions** that allow statements and declarations to be grouped together to perform a specific algorithm or task. A function can be used multiple times in one program or used in different programs.

A function can also navigate through different files that make up a program. Functions are a very important part of the C language. The C language is widely used to write APIs ("Application Programming Interfaces") such as the APIs of Google's TensorFlow software for Artificial Intelligence, APIs for computer graphics (Vulkan, OpenGL), etc. The C language is used to write these APIs because of its versatility and speed of execution. In general, APIs are Functions that transfer their results to the outside by using pointer variables as parameters. This can certainly be done efficiently in the C language.

We will use the following analogy to describe a function:

A function is a Transformer that accepts parameters (data of a certain Type) to execute a task or algorithm and that once performed, the task or algorithm is transformed into a single **result** or **return** of a **Type** accepted in the C language, with certain exceptions.

The following figure graphs a function:



The figure above represents the way a function is most commonly used in a program, that is, when it receives parameters and returns or transforms into a variable of a data type.

The **BODY** of the function is that part of the definition enclosed in curly brackets and represents all the statements and declarations that make up (algorithm) what the function executes.

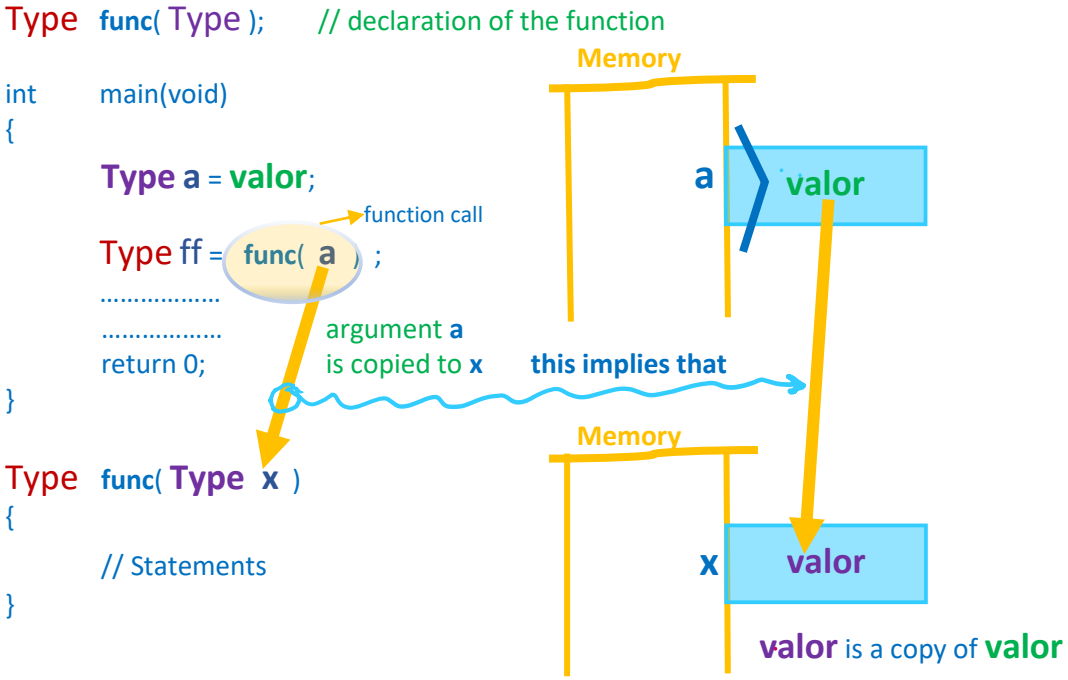
If a function does not return anything, the **void** type is used for the return. In this case, the function is not transformed into anything when executed and its effects depend on the statements used in the definition.
Example:

```
void   func2(int x)
{
        printf("%d\n", x);
}
```

When executed, this function prints the value of the variable x on the screen. It does not return anything.
The call to a function like func2(...) is simply:

```
func2(34); // as any statement
           // prints the value 34.
```

An argument in a function call is always copied to a parameter of the same type in the function definition.
This is explained in the following diagram:



Example: If **Type** is a float, then **valor** is a decimal number that is copied into **x**. If **Type** is a type-to-pointer, then **a** is a pointer and **valor** is an address (of some object) that is copied into **x**. Importantly, the Type of the argument **a** must be equal to the Type of the parameter **x**.
If the argument **a** is directly the address of a variable such as &variable, then it is considered a constant and is copied directly into the **x** parameter.
If a were a numeric value, not a variable, it is copied directly into **x**.

Can a function be used as a parameter of another function?

The answer to this question is no and yes. It is not possible directly but converting a pointer-to-function variable to a new pointer-to-function Type is feasible.

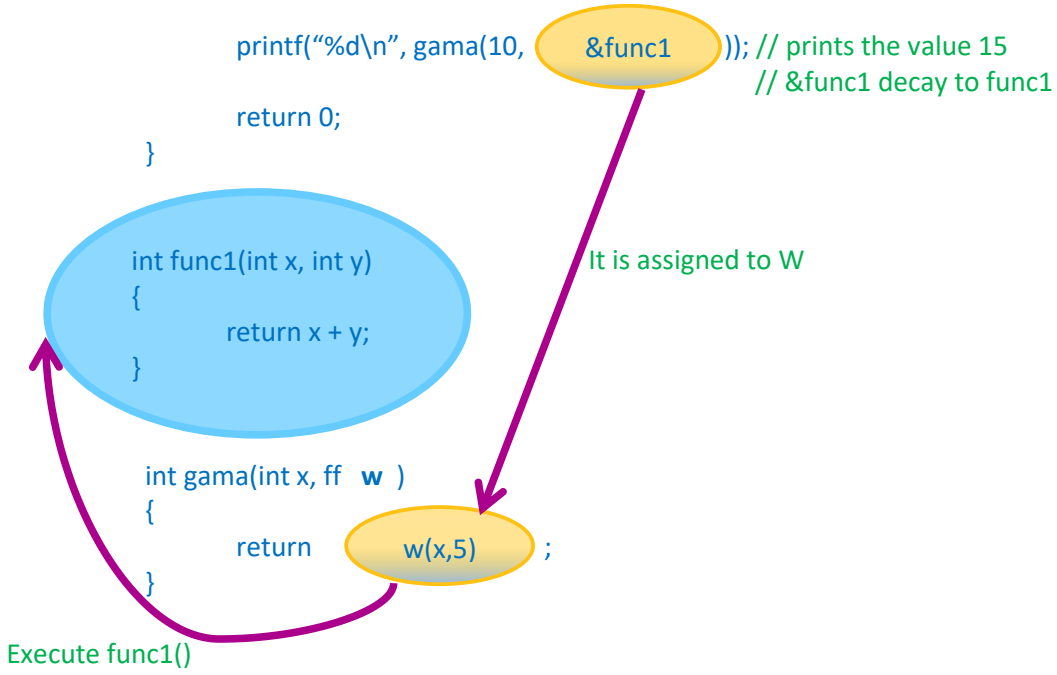
The statement:

```
typedef    int  ( *ff )( int ,int);
```

makes **ff** a **pointer-to-function Type**. If **ff** is a Type, then it is possible to declare a pointer-to-function variable of this type.

The above program is modified to include a function as a parameter of another function:

```
#include <stdio.h>

typedef int (*ff)(int, int);  // now ff is a TYPE pointer-to-function

int func1(int, int);
int gama(int, ff);     // gama includes as parameter the type ff

int main(void)
{
        printf("%d\n", gama(10, &func1 )); // prints the value 15
                                           // &func1 decay to func1
        return 0;
}

int func1(int x, int y)
{
        return x + y;
}

int gama(int x, ff  w  )
{
        return  w(x,5) ;
}
```



5.15.8 Anagram.

Enter a word per keyboard between 5 and 15 characters of the English alphabet. Then enter three space-separated words, where each of the three words must have exactly the same number of characters as the first word. Your program should detect if any of the three words entered do not meet the required number of characters and then prompt for a new entry of three words.

Write a function .... anagram(...) that tells if any of the last three words entered is an anagram of the first word entered on the keyboard.

Print all words that are anagrams of the first word.

An anagram is a permutation of the characters of one word to build another.

Example: **agranda – granada**. An anagram is an anagram if two words have the same characters arranged in a different way.

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

// determine what each of the four functions declared below does.
_Bool checkPalabras(char*);
_Bool countLetras(char*,int*);
_Bool igualCantidadLetras(char*,int);
void  anagrama(char*,char*,int);

int main(int argc, char *argv[])
{
   int count = 0;
          char palabraM[50];            // reserve space for first word.
          char tresPalabras[150];     // reserve space for three words.
          do{
          printf("Enter first word between 5 and 15 characters = ");  // enter first Word.
          scanf(" %s", palabraM);}while(!countLetras(palabraM, &count));
          printf("%s\n",palabraM);
          printf("First word =%d\n",count);
```