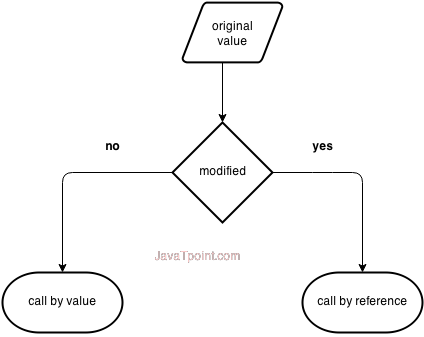
**What are differences between Call by Value and Call by Reference**

There are two ways to pass arguments/parameters to function calls -- *call by value* and *call by reference*. The major difference between call by value and call by reference is that in call by value a copy of actual arguments is passed to respective formal arguments. While, in call by reference the location (address) of actual arguments is passed to formal arguments, hence any change made to formal arguments will also reflect in actual arguments.

|  |  |
| --- | --- |
| Difference between *call by value* and *call by reference* | |
| **call by value** | **call by reference** |
| In *call by value*, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. | In *call by reference*, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function. |
| In call by value, actual arguments will remain safe, they cannot be modified accidentally. | In *call by reference*, alteration to actual arguments is possible within from called function; therefore the code must handle arguments carefully else you get unexpected results. |

There are two ways to pass value or data to function in C language: *call by value* and *call by reference*. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in c language one by one.

## Another difference between call by value and call by reference in c

|  |  |  |
| --- | --- | --- |
|  | **Call by value** | **Call by reference**/Pointer/Address |
| 1 | A copy of value is passed to the function | An address of value is passed to the function |
| 2 | Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected outside the function also |
| 3 | Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |
|  | #include<stdio.h>  **void** interchange(**int** number1,**int** number2)  {  **int** temp;  temp = number1;  number1 = number2;  number2 = temp;  }  **int** main() {  **int** num1=50,num2=70;  interchange(num1,num2);  printf("\nNumber 1 : %d",num1);  printf("\nNumber 2 : %d",num2);  **return**(0);  }  Output :  Number 1 : 70  Number 2 : 50 | #include<stdio.h>  **void** interchange(**int** \*num1,**int** \*num2)  {  **int** temp;  temp = \*num1;  \*num1 = \*num2;  \*num2 = temp;  }  **int** main() {  **int** num1=50,num2=70;  interchange(&num1,&num2);  printf("\nNumber 1 : %d",num1);  printf("\nNumber 2 : %d",num2);  **return**(0);  }  Output :  Number 1 : 70  Number 2 : 50 |

**What is meant by recursion in C?**

A function that calls itself is known as a recursive function. And, this technique is known as recursion. **It** is a programming technique that allows the programmer to express operations in terms of themselves. In **C**, this takes the form of a **function** that calls itself. A useful way to think of **recursive functions** is to imagine them as a process being performed where one of the instructions is to "repeat the process".

Recursive programs are generally slower than non recursive programs because it needs to make a function call so the program must save all its current state and retrieve them again later. This consumes more time making recursive programs slower. Recursive programs requires more memory to hold intermediate states in a stack. Non recursive programs don't have any intermediate states, hence they don't require any extra memory

**Recursion** is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a **recursive** call of the function.void **recursion**() { **recursion**(); /\* function calls itself \*/ } int main(){ **recursion**(); }