Tutorial Link https://codequotient.com/tutorials/Parameter passing techniques/59fdda0fe63d6b7fd5debfce

TUTORIAL

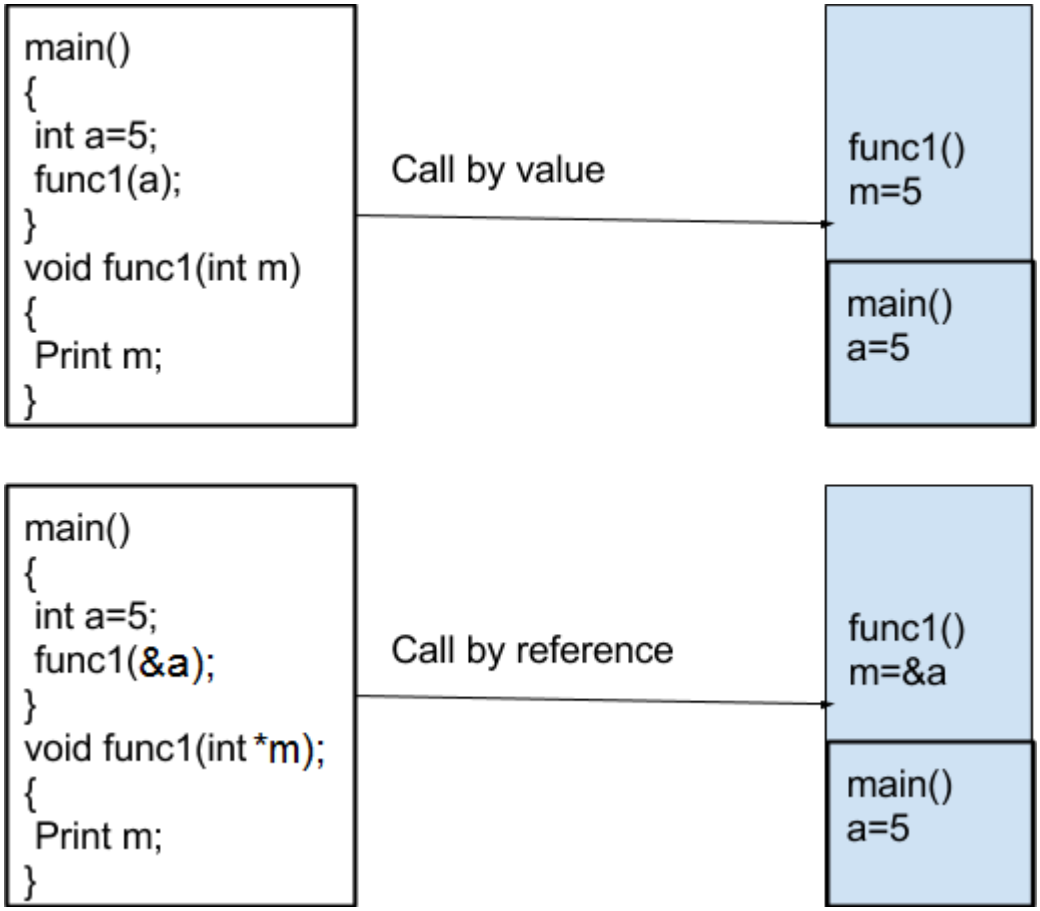
Parameter passing techniques

Chapter

Mainly the parameters passed will take either of these two situations in most modern programming languages,

> Call by Value

> Call by Reference.



In first, the value of actual argument is copied to the formal argument and both have their own existence in memory. Whereas in second only the actual argument stored in memory and formal arguments will refer to the memory locations of actual arguments producing a aliasing kind of effect. So in first the change in formal argument will not be reflected in main program as it is done to local copy only, where in second a change in formal argument will be a change in actual argument, as there is only one copy in memory and both refers to it. Following program illustrate this: -

```c
#include<stdio.h>
void swap(int a, int b)
{
        int tmp;
        printf("In swap() function:\n");
        printf("values before swap a = %d\t and b = %d\n", a, b);
        tmp = a;
    a = b;
```

```
    b = tmp;
        printf("In swap() function:\n");
        printf("values after swap a = %d\t and b = %d\n", a, b);
}

int main() {
        int m = 22, n = 44;
        printf("In main() function:\n");
        printf("Values before swap  m = %d\tand n = %d\n", m, n);
        swap(m, n);             // calling swap function by value
        printf("In main() function:\n");
        printf("Values after swap  m = %d\tand n = %d\n", m, n);
}

```

The output of the above function is: -

```
In main() function:
Values before swap  m = 22      and n = 44
In swap() function:
values before swap a = 22        and b = 44
In swap() function:
values after swap a = 44         and b = 22
In main() function:
Values after swap  m = 22       and n = 44
```

In this program, the values of two variables are being passed by value to the function, so any change done by the function will not reflect in the main function. Whereas if we pass the arguments by reference (which is achieved in C by passing the address instead of values), then the changes will reflect as they are done to the same memory locations. For example: -

```c
#include<stdio.h>

void swap(int *a, int *b)
```

```
{
        int tmp;
        printf("In swap() function:\n");
        printf("values before swap *a = %d\t and *b = %d\n", *a, *b);
        tmp = *a;               *a = *b;
                *b = tmp;
        printf("In swap() function:\n");
        printf("values after swap *a = %d\t and *b = %d\n", *a, *b);
}

int main()
{
        int m = 22, n = 44;
        printf("In main() function:\n");
        printf("Values before swap  m = %d\tand n = %d\n", m, n);
        swap(&m, &n);           // calling swap function by reference
        printf("In main() function:\n");
        printf("Values after swap  m = %d\tand n = %d\n", m, n);
}

```

The output of the above function is: -

```
In main() function:
Values before swap  m = 22      and n = 44
In swap() function:
values before swap a = 22        and b = 44
In swap() function:
values after swap a = 44         and b = 22
In main() function:
Values after swap  m = 44       and n = 22
```

Parameter Passing Order

The order in which arguments are passed to a function when a function call is encountered, is also an important consideration sometimes. There are two possibilities for parameter passing sequence in case of more than one parameters for a function:

(a) Arguments might be passed from left to right.

(b) Arguments might be passed from right to left.

C language follows the second order. For example, Consider the following function call:

```
func1 (x, y, z) ;
```

In this call it doesn't matter whether the arguments are passed from left to right or from right to left. However, in some function calls the order of passing arguments becomes an important consideration. For example:

```
int x = 1;
printf ("%d %d %d", x, ++x, x++ ) ;
```

It appears that this printf( ) would output 1 2 3.

This however is not the case. Surprisingly, it outputs 3 3 1. This is because C's calling convention is from right to left. That is, firstly 1 is passed through the expression x++ and then x is incremented to 2. Then result of ++x is passed. That is, x is incremented to 3 and then passed. Finally, latest value of x, i.e. 3, is passed. Thus in right to left order 1, 3, 3 get passed. Once printf( ) collects them it prints them in the order in which we have asked it to get them printed (and not the order in which they were passed). Thus 3 3 1 gets printed.