Arguments and Function Scope

This lesson deals with the nature of the arguments being passed to a function.

WE'LL COVER THE FOLLOWING

In the previous lesson, we learned how to pass arguments into a function. It is a very simple process, but there are a few things we need to know about how the C++ compiler treats these arguments.

Pass-by-Value

The compiler never actually sends the variables into the function. Instead, **a copy of each argument is made and used in the scope of the function**. This concept is known as **passing-by-value**. Only the values of the arguments are passed into the functions, not the arguments themselves.

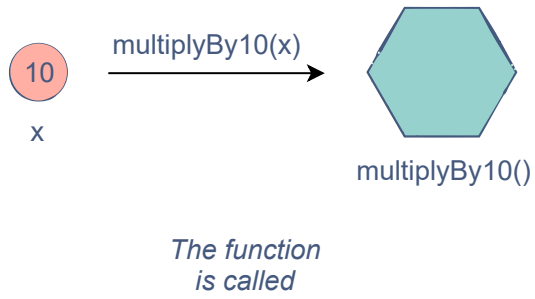
When the function ends, all the argument copies and the variables created inside the function are destroyed forever. Basically, changing the value of an argument inside a function won't change it outside the function.

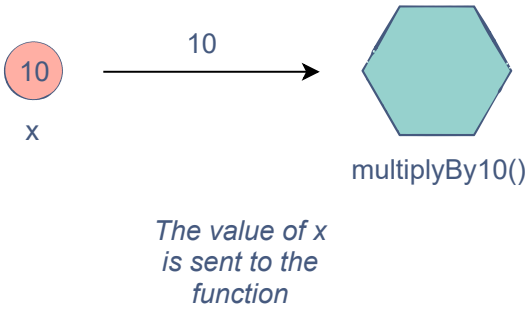
Have a look at the function below, which multiplies its argument by 10:

```
#include <iostream>
using namespace std;

void multiplyBy10(int num){
  num = num * 10;
}

int main(){
  int x = 10;

  cout << "Before function call" << endl;
  cout << "x: " << x << endl;
```

```
  // Multiplying by 10
  multiplyBy10(x);

  cout << "After function call" << endl;
  cout << "x: " << x << endl;
}
```

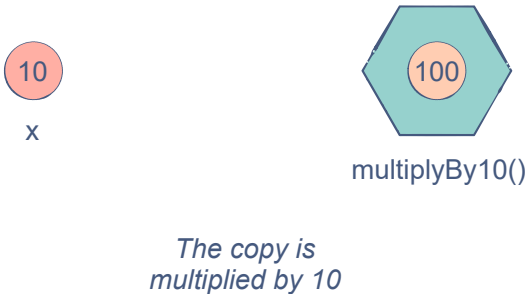
Just as we discussed, the value of `x` did not change outside the function scope. The illustration below will help up understanding this better.

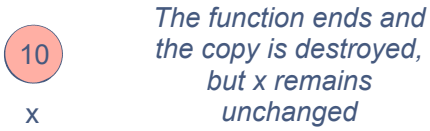




The value of x is sent to the function

A copy of x is used in the function scope





This is normal behavior in C++ but in some cases, we might want a function to manipulate variables outside its scope. Well, there is a solution for that.

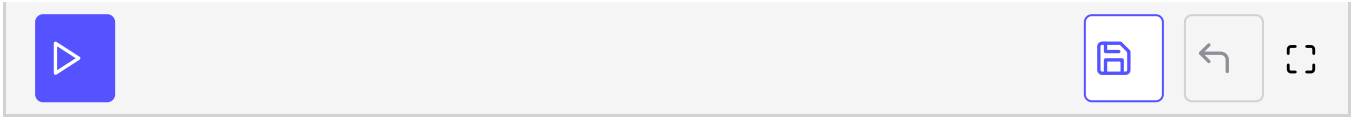
Pass-by-Reference

This is the second approach of passing elements to functions. As we know, every variable is stored at an address in the memory.

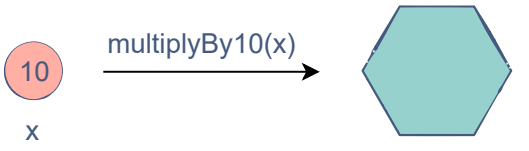
Passing-by-reference means passing the address of a variable as an argument instead of the variable itself. This will ensure that the function manipulates the actual variable.

To access the address of a variable, we need to add the `&` operator before it. Let's refactor the `multiplyBy10` so that we pass `x` by reference.

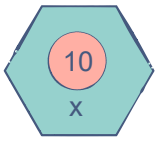
```
#include <iostream>
using namespace std;

void multiplyBy10(int &num){ // num will contain the reference of the
                             // input variable
  num = num * 10;
}

int main(){
  int x = 10;

  cout << "Before function call" << endl;
  cout << "x: " << x << endl;

  // Multiplying by 10
  multiplyBy10(x);

  cout << "After function call" << endl;
  cout << "x: " << x << endl; // The actual value of x is changed!
}
```



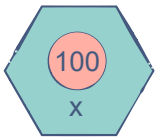
Voila! With a simple `&`, we can allow a function to edit things outside its scope. Let's see what happens when we pass arguments by reference:



The function is called



The value of x is sent to the function



multiplyBy10()

x is multiplied by 10

3 of 4

The function ends, and the value of x has changed

4 of 4

Now, we can decide when we need to pass an argument by value or by

reference. The next lesson will explain how different arguments can affect the same function.