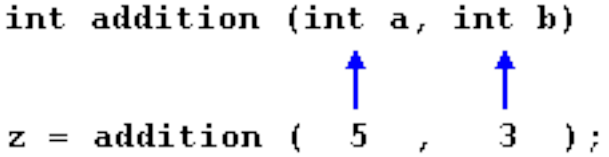
Arguments passed by value and by reference (written by Professor Marina Tanasyuk)

In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

```
1 int x=5, y=3, z;
2 z = addition ( x, y );
```

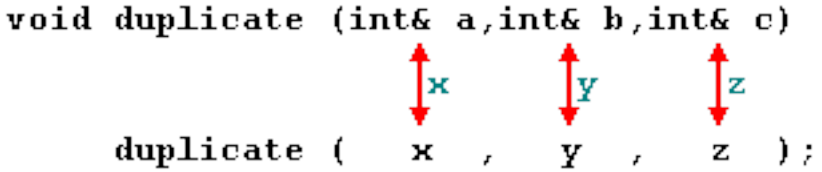
In this case, function addition is passing parameters with values 5 and 3, which are copies of the values of `x` and `y`, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside of it, because x and y themselves were not passed to the function on the call, but only copies of their values at that moment.



In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function `duplicate` is passing the memory addresses of its three arguments, causing the variables used as arguments to actually be modified by the call:

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (`&`) following the parameter type, as in the parameters taken by `duplicate` in the example above.

When a variable is passed *by reference*, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.



In fact, `a`, `b`, and `c` become aliases of the arguments passed on the function call (`x`, `y`, and `z`) and any change on `a` within the function is actually modifying variable `x` outside the function. Any change on `b` modifies `y`, and any change on `c` modifies `z`. That is why when, in the example, function `duplicate` modifies the values of variables `a`, `b`, and `c`, the values of `x`, `y`, and `z` are affected.

Example:

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

void printDigits(int n) {
        while (n > 0) {
                cout << n % 10 << " ";
                n /= 10;
        }
}

void printDigitsRef(int& n) {
        while (n > 0) {
                cout << n % 10 << " ";
                n /= 10;
        }
}

int main() {
        int n;
        cout << "Enter a positive number:  ";
        cin >> n;
        if (n < 0) return 0;

        cout << "Digits are: ";
        printDigits(n);                                          (1)
        cout << endl;
        cout << "The value of n is now: " << n << endl;          (2)

        cout << “Digits are: “;
        printDigitsRef(n);                                       (3)
        cout << endl;
        cout << "The value of n is now: " << n << endl;          (4)

        return 0;
}
```

Sample output:

```
marusya555$ ./a.out
Enter a positive number: 5681
Digits are: 1 8 6 5
The value of n is now: 5681
Digits are: 1 8 6 5
The value of n is now: 0
```

Notes:

When you run this code you'll notice that after you call a function *printDigits(n)* on line (1), it takes a value of *n* and prints digits, and if you print *n* after that, the value of it hasn't change.

But after you call a function *printDigitsRef(n)* on line (3), it uses an actual variable *n* and prints digits of it, but every time the line *n /= 10* runs in the function *printDigitsRef*, the actual variable *n* is changed. If you print *n* after that, the value of it has been changed indeed.

From p.105 of "Schaum's Programming with C++" reference textbook

Passing By Value Versus Passing By Reference

Passing By Value	Passing By Reference
`int x;` The parameter x is a local variable. It is a duplicate of the argument. It cannot change the argument. The argument passed by value may be a constant, a variable, or an expression. The argument is read-only.	`int &x;` The parameter `x` is a local reference. It is a synonym for the argument. It can change the argument. The argument passed by reference must be a variable. The argument is read-write.

A common situation where reference parameters are needed is where the function has to return more than one value. It can only return one value directly with a `return` statement. So if more than one value must be returned, reference parameters can do the job.