FUNCTIONS  (created by professor Marina Tanasyuk)

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

Where:

- **type** is the type of the value returned by the function.

- **name** is the identifier by which the function can be called.

- **parameters** (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: `int x`), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.

- **statements** is the function's body. It is a block of statements surrounded by braces `{ }` that specify what the function actually does.
Let's have a look at an example:

```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b) {
    int r;
    r=a+b;
    return r;
}

int main () {
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
}
```

The result is 8

This program is divided in two functions: `addition` and `main`.

Remember that no matter the order in which they are defined, a C++ program always starts by calling `main`. In fact, `main` is the only function called automatically, and the code in any other function is only executed if its function is called from `main` (directly or indirectly).
In the example above, `main` begins by declaring the variable `z` of type `int`, and right after that, it performs the first function call: it calls `addition`. The call to a function follows a structure very similar to its declaration. In the example above, the call to `addition` can be compared to its definition just a few lines earlier:

```plaintext
int addition (int a, int b)

z = addition (  5  ,  3  );
```

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters `a` and `b`, declared for function `addition`.

At the point at which the function is called from within `main`, the control is passed to function `addition`: here, execution of `main` is stopped, and will only resume once the `addition` function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables `int a` and `int b` within the function.

Then, inside `addition`, another local variable is declared (`int r`), and by means of the expression `r=a+b`, the result of `a` plus `b` is assigned to `r`; which, for this case, where `a` is 5 and `b` is 3, means that 8 is assigned to `r`. 
The final statement within the function:

```
return r;
```

Ends function `addition`, and returns the control back to the point where the function was called; in this case: to function `main`. At this precise moment, the program resumes its course on `main` returning exactly at the same point at which it was interrupted by the call to `addition`. But additionally, because `addition` has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended `addition`: in this particular case, the value of the local variable `r`, which at the moment of the `return` statement had a value of 8.

```
int addition (int a, int b)  
8
z = addition (  5 ,   3 );
```

Therefore, the call to `addition` is an expression with the value returned by the function, and in this case, that value, 8, is assigned to `z`. It is as if the entire function call (`addition(5,3)`) was replaced by the value it returns (i.e., 8).

Then `main` simply prints this value by calling:

```
cout << "The result is " << z;
```
A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```cpp
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b) {
    int r;
    r=a-b;
    return r;
}

int main () {
    int x=5, y=3, z;
    z = subtraction (7,2);
    cout << "The first result is " << z << '
';
    cout << "The second result is " << subtraction (7,2) << '
';
    cout << "The third result is " << subtraction (x,y) << '
';
    z = 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '
';
}
```

The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
Math functions

Header `<cmath>` declares a set of functions to compute common mathematical operations and transformations:

- **Trigonometric functions**
- **Exponential and logarithmic functions**
- **Power functions:**
  - `pow(base, power) -> pow(3, 2) = 3 ^ 2 = 9`
  - `sqrt(value) -> sqrt(16) = 4`
- **Rounding and remainder functions:**
  - `ceil(value) -> round up value -> ceil(2.3) = 3`
  - `floor(value) -> round down value -> floor(2.8) = 2`
  - `round(value) -> round to nearest -> round(2.3) = 2`
- **Minimum, maximum, difference functions**
- **Other functions:**
  - `abs(value) -> abs(-4) = 4`
**rand() function**

(included in `<cstdlib>` library)

Returns a pseudo-random integral number in the range between 0 and \( \text{RAND\_MAX} \). \( \text{RAND\_MAX} \) - this value is library-dependent, but is guaranteed to be at least 32767 on any standard library implementation.

A typical way to generate trivial pseudo-random numbers in a determined range using `rand` is to use the modulo of the returned value by the range span and add the initial value of the range:

\[ \text{rand}() \mod v2 + v1 \]

\( v1 \) is the starting point of the range, including (by default is 0)
\( v2 \) is how many numbers should be in the range

Examples:

\( v1 = \text{rand}() \mod 100; // v1 \) in the range 0 to 99
\( v2 = \text{rand}() \mod 100 + 1; // v2 \) in the range 1 to 100
\( v3 = \text{rand}() \mod 30 + 1985; // v3 \) in the range 1985-2014