

## **Lecture 4: Functions**

Curtin FIRST Robotics Club (FRC) Pre-season Training

---

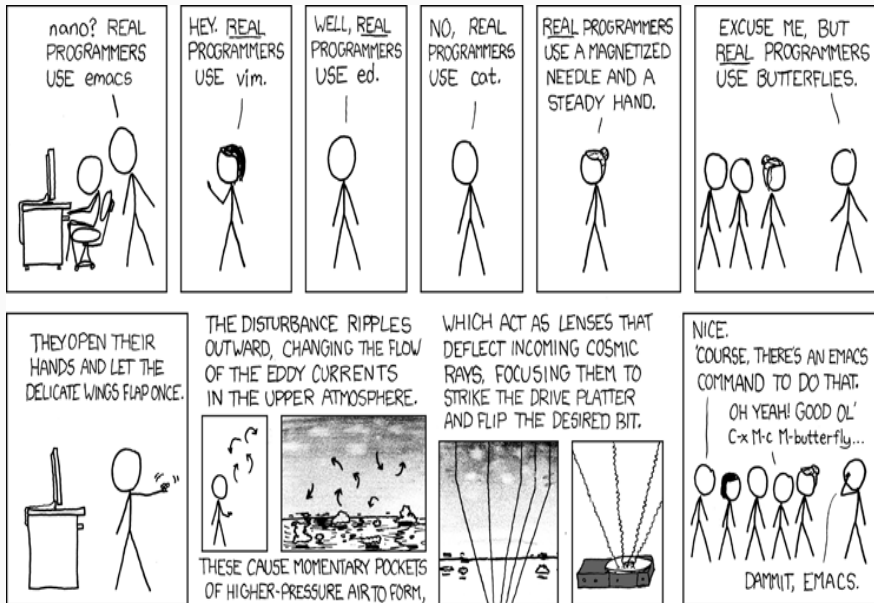
Scott Day

265815F@curtin.edu.au

November 12, 2016

Curtin University

# Insert Mandatory Programming Joke



# Table of contents

1. Function Justification
2. Function Definition
3. Example of Function Definition, Declaration and Call
4. Function Header and Body
5. Function Declaration
6. Function Call and Execution
7. Function Arguments
8. Passing by Value or Reference
9. Recursion - just to f\*\*\* with you

## Function Justification

---

In C++ we can subdivide the functional features of a program into blocks of code known as functions. In effect these are subprograms that can be used to avoid the repetition of similar code and allow complicated tasks to be broken down into parts, making the program modular.

Until now you have encountered programs where all the code (statements) has been written inside a single function called `main()`. Every executable C++ program has at least this function. In the next sections we will learn how to write additional functions.

## Function Definition

---

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:

```
1 returnType functionName(type parameter1, type parameter2, ...)  
2 {  
3     statements;  
4 }
```

Where:

<b>return-value-type</b>	Is the type of the value returned by the function.
<b>function-name</b>	Is the identifier by which the function can be called.
<b>parameters</b>	Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.
<b>statements</b>	Is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.





## **Example of Function Definition, Declaration and Call**

---

## Example 1 - Addition

- A very basic example of using a function
- Note: Both examples do the exact same thing.

```
1 // Function Example
2 #include <iostream>
3 using namespace std;
4
5 int addition(int a, int b)
6 {
7     int result;
8
9     result = a + b;
10
11     return result;
12 }
13
14 int main()
15 {
16     int temp;
17
18     temp = addition(5, 3);
19     cout << "The result is: " << temp << endl;
20
21     return 0;
22 }
```

```
1 // Function Example
2 #include <iostream>
3 using namespace std;
4
5 int addition(int a, int b)
6 {
7     return a + b;
8 }
9
10 int main()
11 {
12     cout << "The result is: " << addition(5, 3) <<
13         << endl;
14
15     return 0;
16 }
```

## Example 2 - Factorial

Let us first look at an example of a program written entirely with the function `main()` and then we will modify it to use an additional **function call**.

We will illustrate this with a program to calculate the factorial ( $n!$ ) of an integer number ( $n$ ) using a for loop to compute:

$$n! = 1 \times 2 \times 3 \dots (n-2) \times (n-1) \times n$$

## Example 2 - Factorial

```
1 // Program to calculate factorial of a number
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int ii, number = 0, factorial = 1;
8
9     // User input must be an integer number between 1 and 10
10    while(number < 1 || number > 10)
11    {
12        cout << "Enter integer number (1-10) = ";
13        cin >> number;
14    }
15
16    // Calculate the factorial with a FOR loop
17    for(ii = 1; ii <= number; ii++)
18    {
19        factorial = factorial * ii;
20    }
21
22    // Output result
23    cout << "Factorial = " << factorial << endl;
24    return 0;
25 }
```

Even though the program is very short, the code to calculate the factorial is best placed inside a function since it is likely to be executed many times in the same program.

## Example 2 - Factorial

```
1 // Program to calculate factorial of a number with
  ↳ function call
2 #include <iostream>
3 using namespace std;
4
5 // Function declaration (prototype)
6 int factorial(int number);
7
8 int main()
9 {
10     int number = 0, result;
11
12     // User input must be an integer number between
  ↳ 1 and 10
13     while(number < 1 || number > 10)
14     {
15         cout << "Integer number = ";
16         cin >> number;
17     }
18
19     // Function call and assignment of return value
  ↳ to result
20     result = factorial(number);
21
22     // Output result
23     cout << "Factorial = " << result << endl;
24     return 0;
25 }

27 // An integer, number, is passed from caller
  ↳ function.
28 int factorial(int number)
29 {
30     int ii, factorial = 1;
31
32     // Calculate the factorial with a FOR loop
33     for(ii = 1; ii <= number; ii++)
34     {
35         factorial = factorial * ii;
36     }
37
38     return factorial; // This value is returned to
  ↳ caller
39 }
40
```

## Example 2 - Factorial Discussion

Three modifications have been made to incorporate a function:

- The **declaration** of the function above `int main()` .  
The declaration (A.K.A the prototype) tells the compiler about the function and the type of data it requires and will return on completion.
- The **function call** in the main body of the program determines when to branch to the function and how to return the value of the data computed back to the main program.
- The **definition** of the function `int factorial(int number)` below the main program.  
The definition consists of a **header** which specifies how the function will interface with the main program and a **body** which lists the statements to be executed when the function is called.

## Remember the Cuteness - A nose so sharp it could fly



## Function Header and Body

---



The function is defined below the body of `int main()`. The **header** in this example:

```
1 int factorial(int number);
```

indicates that the `factorial()` function expects to be **passed** an integer value (the parameter type) from the main body of the program and that the value passed will be stored locally in a variable named `number` (the formal parameter name).

The `return value` type of the function is also `int` in this example, indicating that at the end of executing the body of the function, an integer value will be returned to the statement in which the function was called. Functions which do not return a value have `return value` type `void`.

The **body** of the function computes the factorial of a number in exactly the same way as in the example with only a `main()` function.

The execution of the function terminates with a `return` statement:

```
1      return factorial;
```

which specifies that the value stored in the function variable `factorial` should be passed back to the calling function.



## Function Declaration

---

## Function Declaration

Every function has to be **declared** before it is used. The declaration tells the compiler the name, return value type and parameter types of the function.

In this example the declaration:

```
1 int factorial(int number);
```

tells the compiler that the program passes the value of an integer to the function and that the return value must be assigned to an integer variable. The declaration of a function is called its **prototype**, which means the "first" time the function is identified to your program.

The function prototype and the function definition must agree exactly about the return value type, function name and the parameter types. The function prototype is usually a copy of the function header followed by a semicolon to make it a declaration and placed before the main program in the program file.

## Function Call and Execution

---

## Function Call and Execution

The function definition is entirely passive. By itself it does nothing unless instructed to execute. This is done by a statement in the main program called the **function call**.

For example the statement:

```
1 result = factorial(number);
```

calls the function `factorial()` and passes a copy of the value stored in the variable, `number`. When the function is called, computer memory is allocated for the parameter, `number` and the value passed is copied to this memory location. Memory is also allocated to the (local) variables `factorial` and `ii`. The statements of the function are then executed and assign a value to the variable `factorial`. The return statement passes this value back to the calling function. The memory allocated to the parameters and local variables is then destroyed. The value returned is assigned to the variable on the left-hand side, `result`, in the expression used to call the function. The net effect of executing the function in our example is that the variable `result` has been assigned the value of the factorial of `number`.

## Example Execution

A function can be called any number of times from the same or different parts of the program. It can be called with different parameter values (though they must be of the correct type).

For example the following fragment of code can be used to print out the factorials of the first 10 integers:

```
1 for(ii = 1; ii <= 10; ii++)
2 {
3     result = factorial(ii);
4     cout << ii << "!" = " << result << endl;
5 }
```

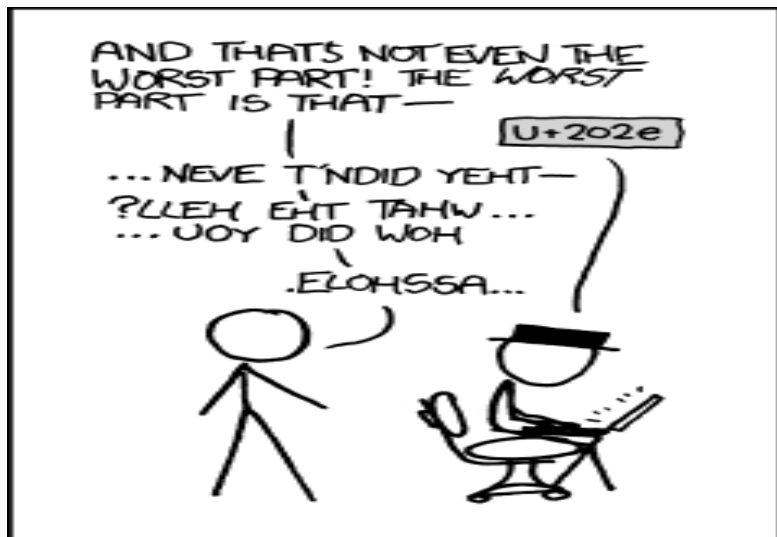
and

```
1 binomialCoefficient = factorial(n) / (factorial(k) * factorial(n - k));
```

can be used to compute the binomial coefficient  $\frac{n!}{k!(n-k)!}$



Programming memes for those who don't know what a binomial coefficient is - all good I forgot aswell



## Function Arguments

---

t

t

t

## Passing by Value or Reference

---

t

t



t

Recursion - just to f\*\*\* with you

---

t

t

t

