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

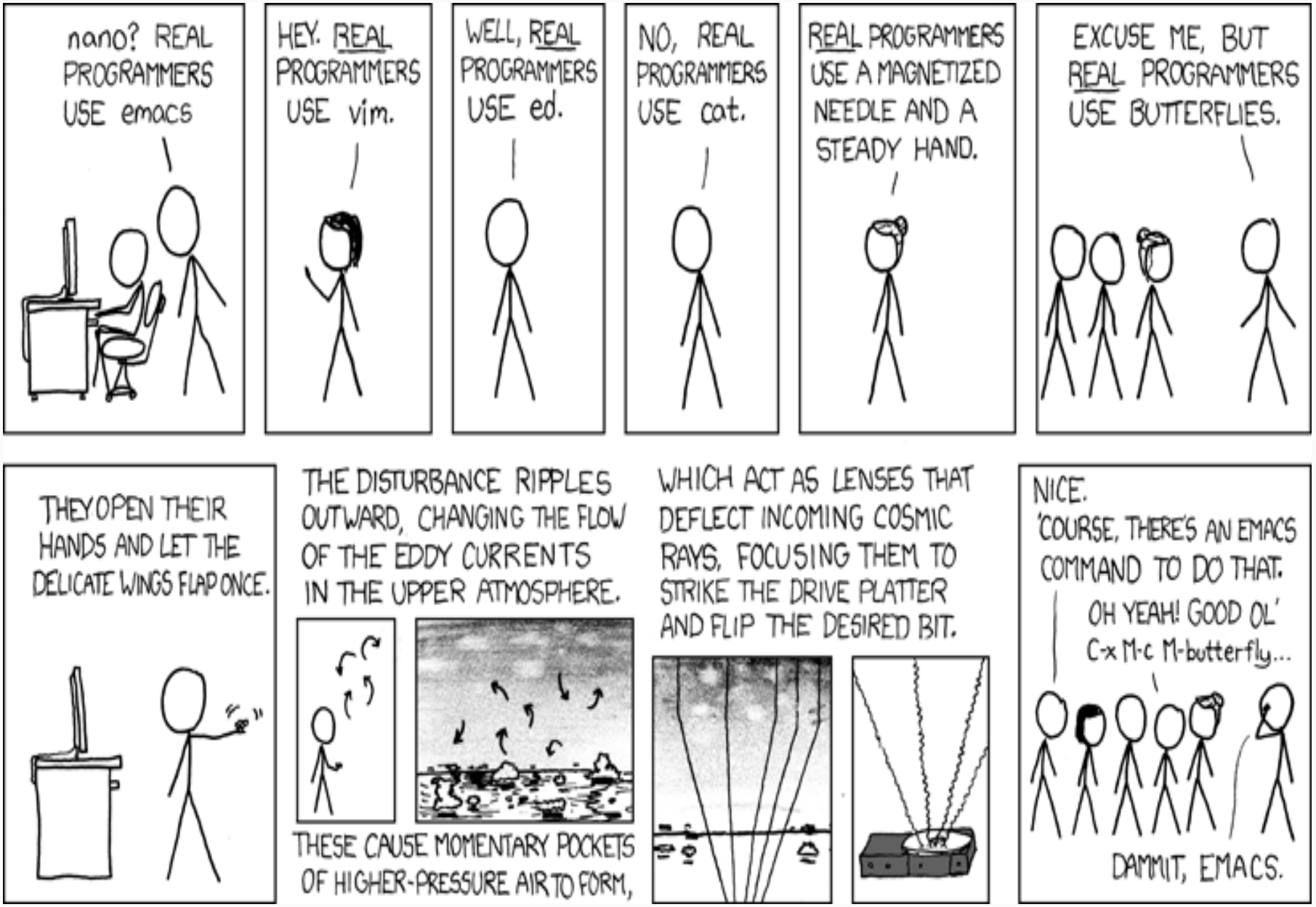


Table of contents

Function Justification

Functions

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:

```
returnValueType functionName(type parameter1, type parameter2, ...)
{
    statements;
}
```

Where:

return-value-type Is the type of the value returned by the function.

function-name Is the identifier by which the function can be called.

parameters Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.

statements Is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Cute Dog

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.

```cpp
// Function Example
#include <iostream>
using namespace std;

int addition(int a, int b)
{
    int result;

    result = a + b;

    return result;
}

int main()
{
    int temp;

    temp = addition(5, 3);
    cout << "The result is: " << temp << endl;

    return 0;
}
```

```cpp
// Function Example
#include <iostream>
using namespace std;

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

int main()
{
    cout << "The result is: " << addition(5, 3) <<
    ↪    endl;

    return 0;
}
```

Example 2 - Factorial

Let us first look at an example of a program writen 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 factoriall (n!) of an integer number (n) using a for loop to compute:

`n! = 1 x 2 x 3...(n-2) x (n-1) x n`

Example 2 - Factorial

```cpp
// Program to calculate factorial of a number
#include <iostream>
using namespace std;

int main()
{
    int ii, number = 0, factorial = 1;

    // User input must be an integer number between 1 and 10
    while(number < 1 || number > 10)
    {
        cout << "Enter integer number (1-10) = ";
        cin >> number;
    }

    // Calculate the factorial with a FOR loop
    for(ii = 1; ii <= number; ii++)
    {
        factorial = factorial * ii;
    }

    // Output result
    cout << "Factorial = " << factorial << endl;
    return 0;
}
```

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

```cpp
// Program to calculate factorial of a number with function call
#include <iostream>
using namespace std;

// Function declaration (prototype)
int factorial(int number);

int main()
{
    int number = 0, result;

    // User input must be an integer number between 1 and 10
    while(number < 1 || number > 10)
    {
       cout << "Integer number = ";
       cin >> number;
    }

    // Function call and assignment of return value to result
    result = factorial(number);

    // Output result
    cout << "Factorial = " << result << endl;
    return 0;
}
```

```cpp
// An integer, number, is passed from caller function.
int factorial(int number)
{
    int ii, factorial = 1;

    // Calculate the factorial with a FOR loop
    for(ii = 1; ii <= number; ii++)
    {
        factorial = factorial * ii;
    }

    return factorial; // This value is returned to caller
}
```

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

Header

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.

Body

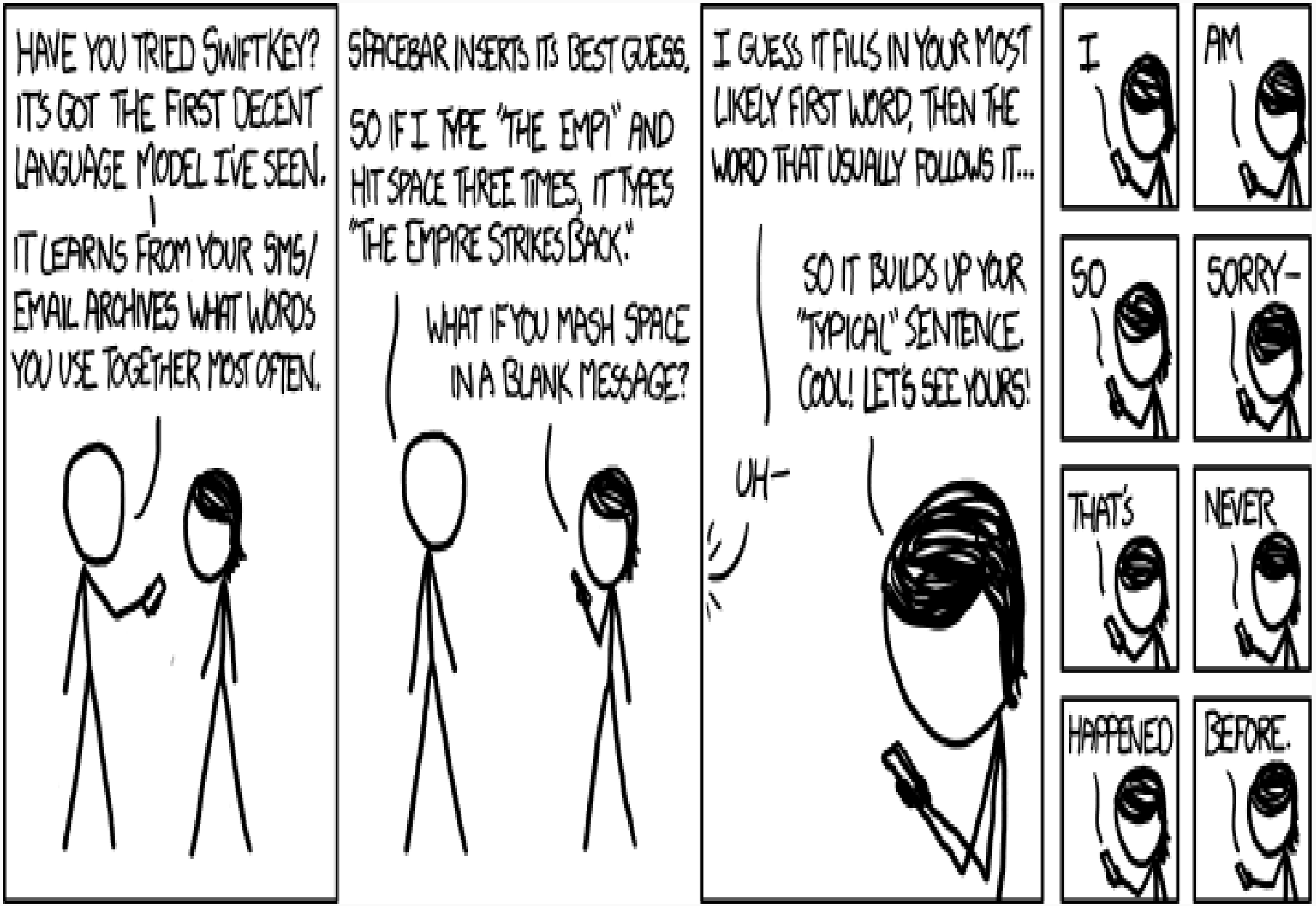
The **body** of the funcction computes the factorial of a number in exactly the same way as in the example with only a `main()` function.

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.

xkcd Programming Memes



Function Declaration

Function Declaration

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

In this example the declaration:

```
1 int fatorial(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 ccalled 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 usally 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:

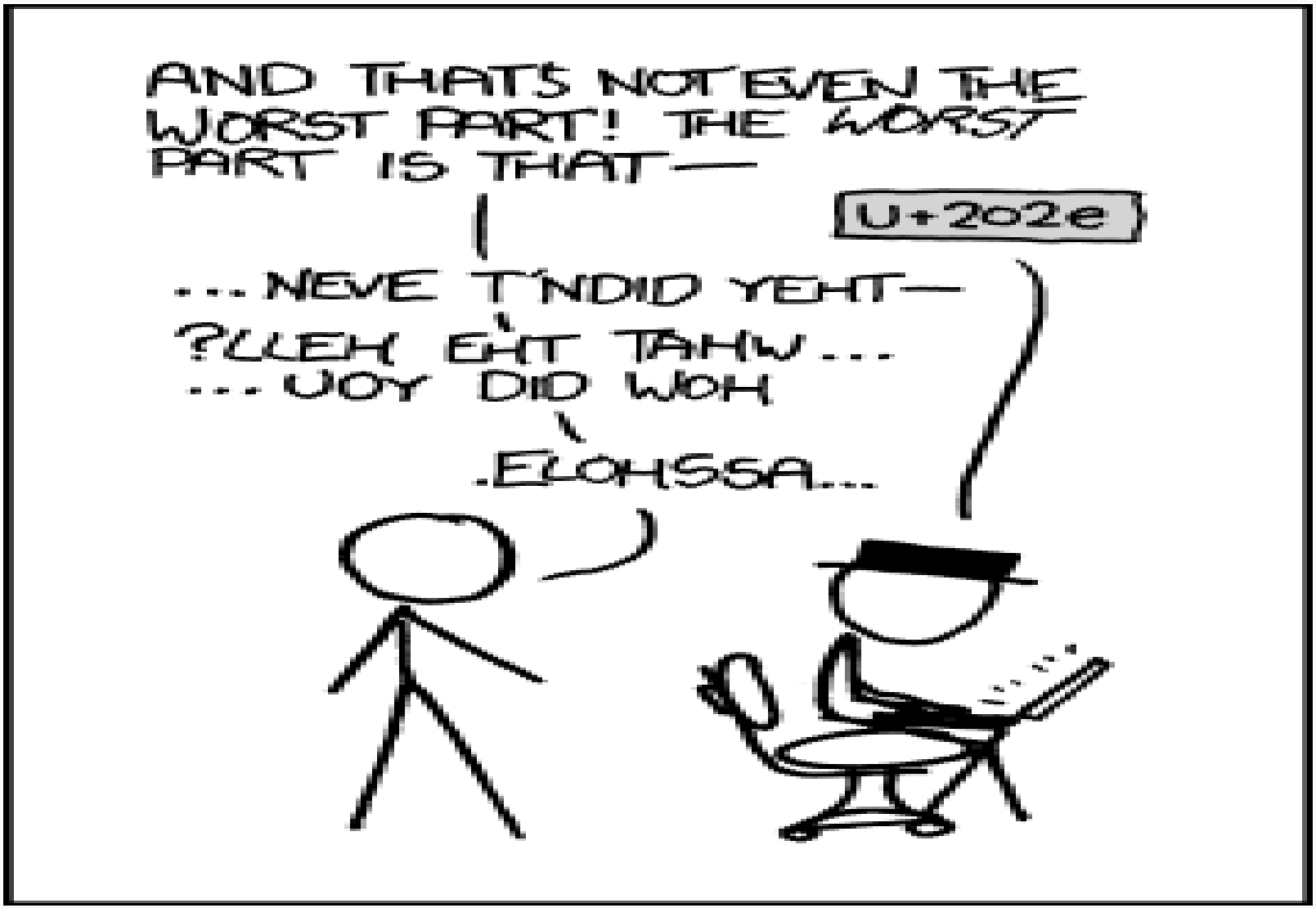
```
for(ii = 1; ii <= 10; ii++)
{
    result = factorial(ii);
    cout << ii << "! = " << result << endl;
}
```

and

```
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

Function Arguments

The names of variables in the statement calling the function will not in general be the same as the names in the function definition, although they must be of the same type.

We often distinguish between the **formal paremeters** in the function definition (e.g. `number`) and the **actual parameters** for the values of the variables passed to the function (e.g. number in the example above) when it is called.

Function arguments (actual parameters) an include constants and mathematical expressions. For example the following statement assigns the value 24 to the variable result.

```
1  result = factorial(4);
```

The function arguments can also be functions that return a value, although this makes the code difficult to read and debug.

Cute Dawg

Passing by Value or Reference

Passing by Value

There are two ways to pass values to functions. Up to now we have only looked at examples of **passing by value**. In the `passing by value` way of passing parameters, a copy of the variable is made and passed to the function.

Changes to that copy do not affect the original variable's value in the calling function.

This prevents the accidental corrupting of variables by functions and so is the preferred method for developing ccorrect and reliable software systems.

One disadvantage of `passing by value` however, is that only a single value can be returned to the caller. If the function has to modify the value of an argument or has to return more than one value, then another method is required.

Passing by Reference

An alternative uses **passing by reference** in which the function is told where in memory the data is stored (i.e. the function is passed the memory address of the variables).

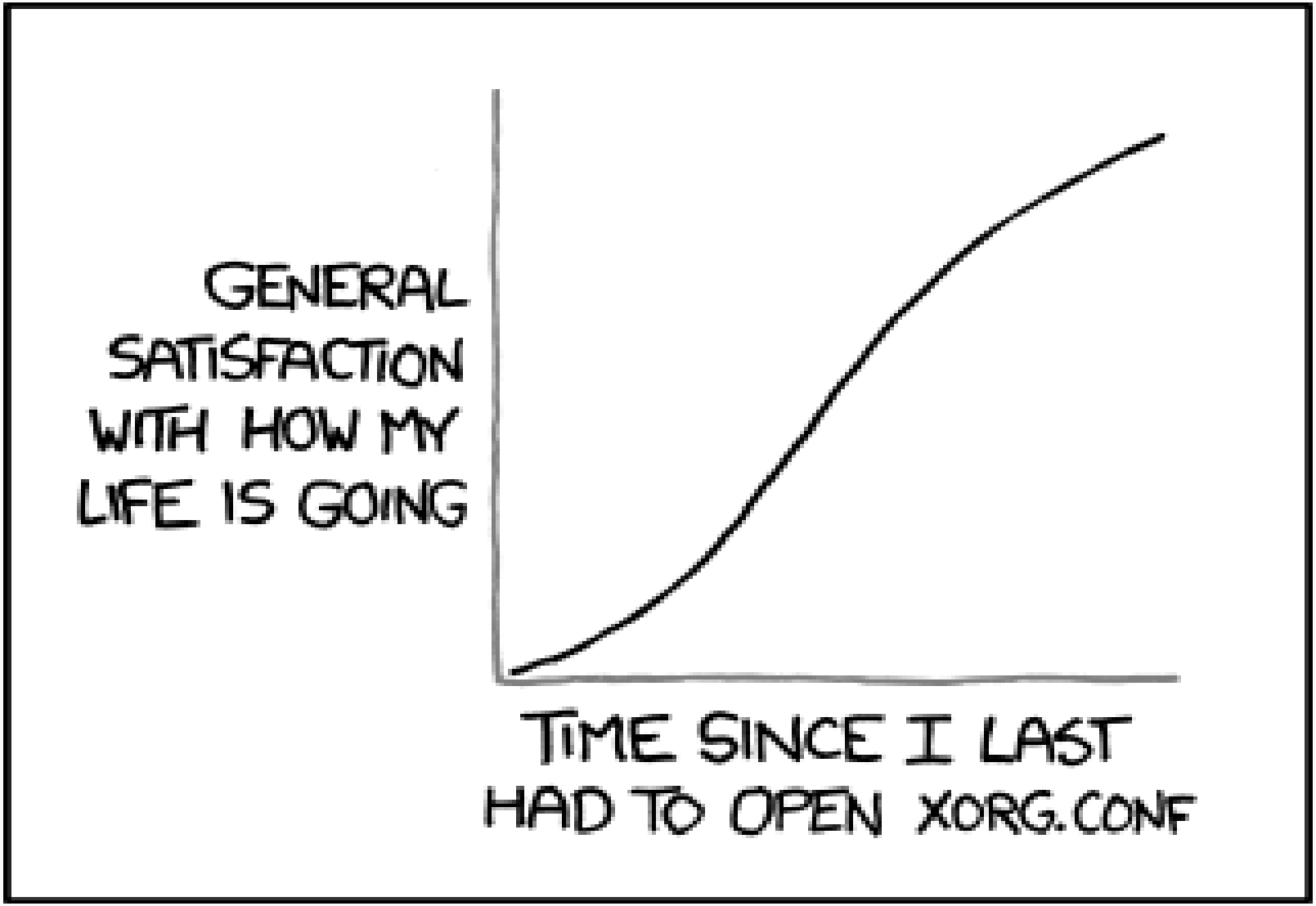
In passing the address of the variable we allow the function to not only read the value stored by also to change it. On the other hand, by passing the value by name we simply let the function know what the value is.

To indicate that a function parameter is `passed by reference` the symbol & is placed next to the variable name in the parameter list of the function definition and prototype (but **not** the function call).

For example:

```
1  void swap(float &x, float &y)
```

When xkcd jokes hit home



References I