Common errors and tips for coding in C

Part 2 Computational Physics

April 21, 2016

THE UNIVERSITY OF
MELBOURNE

1 General good practice

The main issues when it comes to good coding practice are:

1. **Semi-colons:**

   A very common bug that arises for new C programmers is missing semi-colons. Always add these as you code, don't ever leave these with the thought of coming back and adding them later. As soon as you finish a line of code add the semi colon. You don't finish a paragraph and then dot your i's, do it as you go!

2. **Closing brackets:**

   Similar to above always add a closing bracket every time you open one. A much harder bug to find is a miss match in the number of braces and which closing brace corresponds to which opening one. Your main, functions, loops and if statements should all have opening and closing curly braces before they have any contents!

3. **Formatting and indenting** :

   Although it might seem picky, make sure you properly format your code as you go. It is not OK to do it all later. It might seem tedious initially, but you have to maintain a good practice. This isn't just so it looks nice, this is vital for quick and easy reading of the code, which is important when debugging.

   The main thing is to properly indent. This may seem trivial, but as your code gets more complicated, and it will, it will get extremely messy and hard to debug.

   Anything inside a function or the main should be indented by a couple of spaces or a tab. Anything inside a loop or `if` statement indented again. If you have an `if`-statement inside a double `for`-loop inside a function, you should have the first `for` loop indented once, the second once again (twice from the left edge), the `if` statement once again(three times from the left edge) and the contents of the `if` statements a final time (four times from the left edge). This means that at a glance we can see what bit of code belongs to which `if`-statement/loop/function and much easier debugging when checking for mismatched curly braces.

   Check any of the example codes (available in the course notes) as a guide to proper indenting.

2 Non-exhaustive list of common mistakes

In no particular order, the following are some easily overlooked areas you should start to look out for when coding:

1. **Not including the `main`:**

   The `main` is always needed otherwise the code won't run. When the C program executes, it executes the contents of `main` line by line. If no `main` is included, nothing will happen!

2. **Putting code outside of the `main` and other functions:**

   Any code outside of the `main` or any functions will not be run (or cause a crash). The `main` will contain all the code that is run, and any calls to any functions. The only code outside of the `main` should be variables that exist across the whole program and functions.

3. **Not initializing variables:**

   A common cause of a program not running as it should is when variables aren't properly initialized (assigned a starting value). Variables *can* be created without assigning a value, but this should only be done when you are sure that it *will* be assigned a variable before you use it in any arithmetic or in function arguments.

4. **Providing the wrong scope for variables:**

   Scope refers to where variables 'live'. If a variable is created inside a loop, it will only 'exist' inside that specific loop and sub-loops. Trying to access it from outside the loop will either cause the code to not compile, or will call a different variable of the same name, if it exists. The same applies to functions and the `main`; a variable created inside the `main` is only accessible inside the `main`, a variable created inside a function is only accessible inside that function.

5. **Not double-checking arguments and returning variables:**

   The only way to access a variable in a different scope is to pass it as an argument to a function or for a function to return it in a function call. When a variable is passed as a function argument, a copy of that variable is created and exists while that function runs, and is deleted when the function finishes. When a variable is returned, a copy is made and the copy is passed to where the function is called.

6. **Too many global variables:**

   A global variable is a variable that is created outside of the `main` or is created using the `#define` keyword (both usually done at the start of the program, right after the `#include` statements). These exist anywhere in the current program. A common mistake is to define a lot of variables in this way. Global variables should be reserved for cases where the logic requires it, or in most cases, for constants whose value is always the same e.g.: gravity, pi ...

7. **Not prototyping functions:**

   When you have a small amount of functions, where they don't call each other, it is OK to define the functions above the `main`. If you put the functions below the `main`, or one of the functions calls another, the best practice is to prototype. This means that you put all the functions below the `main`, and you put the function definitions (basically the first line of the function definition but with the ' {' replaced with a ';') before the main.

8. **Using the wrong operators:**

   (a) '`=`' vs '`==`' :
   `=` is used for assignment.
   `x=3` means that from now on, `x` will evaluate to `3`.
   `x==3` checks for equality; if `x` is `3` then it will return `True` otherwise it will return `False`.
   e.g.: `if(x==3){x=4}` will set `x` to be equal to `4` if `x` is `3`.

   (b) Using '`!`':
   To check if two variables/numbers are not equal, or to convert between a `False` and a `True` we use the '`!`' (the '`not`' keyword does not exist in C). For example `x!=3` will return `True` if `x` is not equal to 3, `!True` will return `False`.

   (c) Inequalities:
   Remember that as well as '`<`' and '`>`' we can check if something is greater or equal to, or less than or equal to, with : '`>=`' and '`<=`'

9. **Not taking advantage of functions:**

   The main reason we use functions is code reuse. If we want some piece of code to be able to run more than once we should put it in a function. This means that if we ever wish to make a change to the logic, we only have to change it once, not every time we use it.

   This is also very useful when we wish to run the same piece of code, with the same logic, with some parameters changed. This is where arguments come

in. By understanding this and properly implementing it, the use of functions becomes invaluable.

10. **Storing function return types incorrectly:**

    A function's return type is how a function passes information back to the point where the function was called. A function of type `double` will return a double, using this we can think of the function as performing an operation. You give it some input arguments, it processes them, and it returns the output. For example for a function that computes `y=mx+c`, you pass it `m`, `x` and `c` and it will spit back `y` (the logic that the function performs is the same regardless of the input arguments, the output is not).

    If we call a function that returns a variable and do not assign it to a variable, the contents of the code will run, but the return value will not be stored in memory. In the case where we want the value (which would be almost all cases where a function returns a value) we must assign it using the `=` operator. For example `double y = straightLine(m,x,c);` will assign the output to our new variable `y`, based on the given input variables.

11. **Not properly incrementing `For` loops:**

    The counter in a `for` loop is incremented after the code within the loop is executed, it is the last thing that happens before the next loop.

    `for(i=0; i<10; i++)` : The first section in the brackets sets the counter's value, the second is a check done at the start of every loop run, the last is the increment that happens AFTER the code in the '{ }' has run.

12. **Functions of type `void`:**

    The `void` key word is used to specify that a function doesn't return any variable. This is used in cases where the function is just a piece of code we wish to run, and there is no useful variable output. For example a function that plots the position vector given `x` and `y` positions of the form `void plotPosition(x,y);` will give no output as a variable, but will still run, in this case plotting a graph (assuming the correct contents of the function definition).

13. **Dealing with doubles and integers:**

    Numbers including a decimal point evaluate as floating point numbers, those without evaluate as integers.

    Be careful when dealing with arithmetic that involves variables of different types. `1.0/2` will evaluate to `0.5`. `1/2` will evaluate to `0`. This is because `1.0` is a double, so dividing by `2` works fine. `1`, on the other hand is an `int`,

so dividing by `2` gives 0 because integers simply drop any numbers past the decimal point (making `0.5` become `0`). Note there is **no rounding**. `3/4` will evaluate to `0` (since `0.75` will become `0`) not `1`.

To be safe, add a `.0` to any number if this could cause a possible issue in arithmetic or if a particular function requires a float (or double), for example in `printf` and in pgplot functions.

14. **Casting:**

    Casting is when you convert a variable of one type to another. An easy way to convert an `int` to a `double` is to multiply it by 1.0. Assigning an `int` to a `double` or a `double` to an `int` should take care of this automatically. You can also do this explicitly by prefacing a number with the type to cast to in brackets e.g. `(double) 3` or `(int) 5.4` although you shouldn't need to worry about this in these labs.

15. **The difference between creating and accessing arrays, and array index:**

    An array is created using the square brackets to specify its size. From this point on, the square brackets are used to access a particular element of the array. An array is created like so:
    `double myArray[3]`.
    Now calling `double myArray[2]` will pick out the last element, in this case returning a double.

    Array indices run from 0 to the array size. The array created by `double myArray[3]` has 3 elements, the first being accessed by `myArray[0]` and the 3rd by `myArray[2]`.

    Another example: `double myMatrix[2][2]` will create a 2-by-2 matrix. Now calling `double myMatrix[2][2]` will **try** to return the element (a double) in the 3rd row and 3rd column. In this case it doesn't exist so the program will crash. If passing the matrix to a function for example, you would pass `myMatrix` as the argument, as this refers to the double array itself, not `myMatrix[2][2]` which refers to a particular element.

16. **Array assignment and equality operations:**

    You cannot simply assign an array to another. We must make a new empty array and loop over the array, individually assigning each element to the corresponding element in the new array.

    The same is true for checking equality. To check if two matrices are equal we must loop over every element and check that every element is equal.

17. **Dealing with arrays and functions:**

    Arrays are not variables in the same sense as the ones we are used to. They are pointers to the memory address where the array is stored. Because of this we cannot simply pass them to a function as an argument to make a copy, and we cannot simply return an array.

    When an array is passed to a function, the memory address is passed. This means that any changes are made to the array that we input, meaning that any changes will remain even after the function has run. If we pass an array to a function, and double all the elements, the next time the array is accessed (even outside of the function), all the elements will be doubled. So, while we cannot return an array, we don't actually need to as any changes can be retrieved. Be aware of this and careful when dealing with arrays and functions.

18. **Not closing a file:**

    A common mistake is not closing a file before the end of the program, this will cause issues when writing to a file. Another common issue is using the same `FILE` variable twice and not closing it between uses. Always close a file as soon as you are done with a particular task, e.g.: open a file to write to it, as soon as everything is written, close the file.

19. **Not having the right data type in `printf` statements:**

    Always check your `printf` statements. If you are printing a double or float, use `%f`, if printing an `int` use `%d`, if printing a char use `%s`. Not being consistent is a common cause of issues.

    Also make sure that you have the right number of arguments in a `printf`. If the number of `%` in the string doesn't match the number of variable arguments, compiling will fail.

20. **Issues with `scanf`:**

    The most common issue with `scanf` is forgetting to add a `&` before the variables you are writing to. This format is because `scanf` is writing to the memory address of the variables directly, not calling assignment operator (remember that `&x` refers to the memory address of `x`, not the variable that it holds). If you have issues when using `scanf` to read a double, use `%lf` instead of `%f` as `scanf` is more discriminating when it comes to variable types.

21. **Difference between `'` and `"`:**

When using a char, you assign using ” for example `char s = 'l'`. When using a string you assign using "" for example `const char * s = "lalala"`. Mixing these up will cause compilation issues.

Acknowledgements

Originally written by Anton Hawthorne, April 2016.