

## Exercises #1

Time to practice!

First, we'll see one self-explanatory exercise (ex1.c) on common arithmetic operations (like addition, multiplication etc.) between:

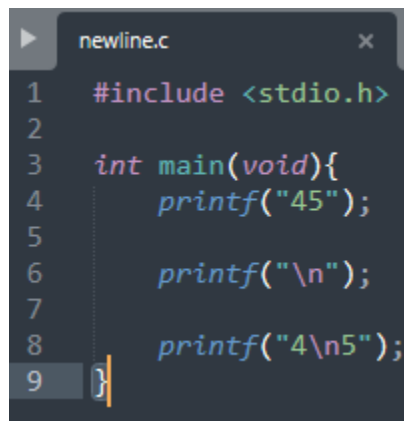
- a) Variables
- b) Variables and arithmetic constants
- c) Arithmetic constants

### *Things that haven't been discussed yet*

#### "\n" – newline character

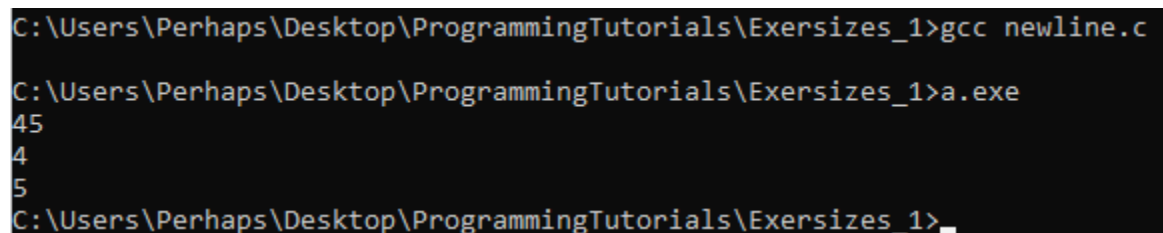
As we've seen before, "\" is the escape character that changes the function of the following character. Therefore, "\n" means go to the next line.

For example:



```
newline.c
1  #include <stdio.h>
2
3  int main(void){
4      printf("45");
5
6      printf("\n");
7
8      printf("4\n5");
9  }
```

When ran, prints out 45 – newline – 4 – newline – 5:

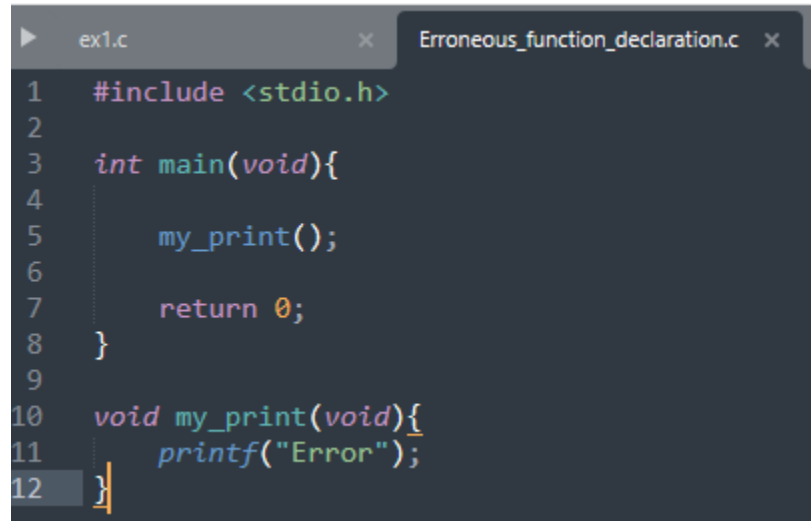


```
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>gcc newline.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>a.exe
45
4
5
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>
```

## Function definition position

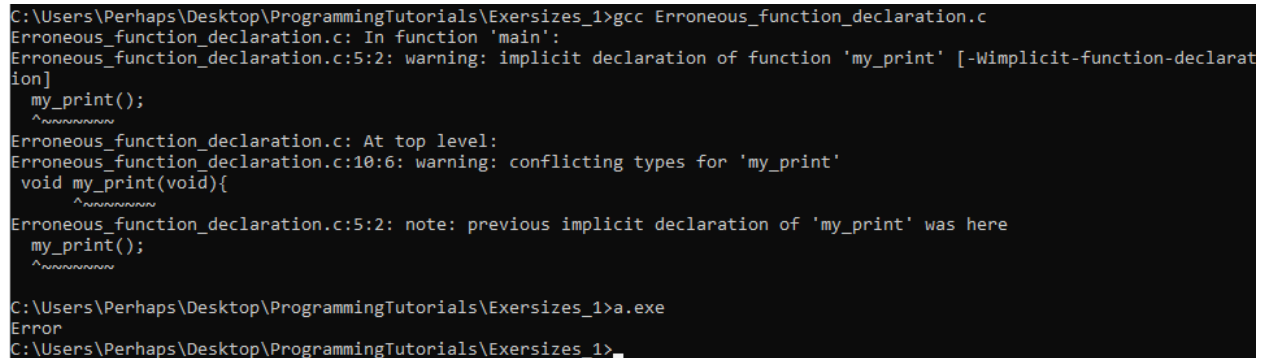
Functions should be defined, *above* the line of code (loc) where they are called for the first time.

For example, this function definition:



```
ex1.c x Erroneous_function_declaration.c x
1  #include <stdio.h>
2
3  int main(void){
4
5      my_print();
6
7      return 0;
8  }
9
10 void my_print(void){
11     printf("Error");
12 }
```

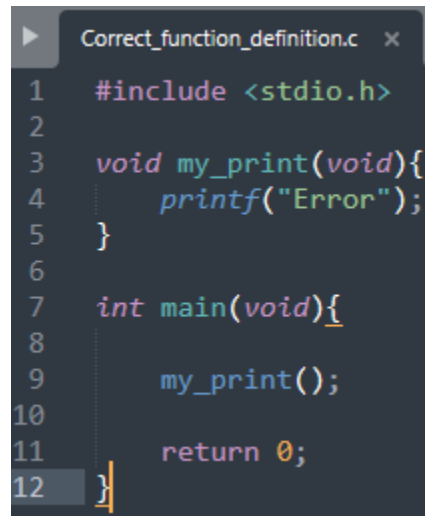
produces a few implicit declaration warnings:



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>gcc Erroneous_function_declaration.c
Erroneous_function_declaration.c: In function 'main':
Erroneous_function_declaration.c:5:2: warning: implicit declaration of function 'my_print' [-Wimplicit-function-declaration]
    my_print();
    ^~~~~~
Erroneous_function_declaration.c: At top level:
Erroneous_function_declaration.c:10:6: warning: conflicting types for 'my_print'
    void my_print(void){
        ^~~~~~
Erroneous_function_declaration.c:5:2: note: previous implicit declaration of 'my_print' was here
    my_print();
    ^~~~~~
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>a.exe
Error
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>
```

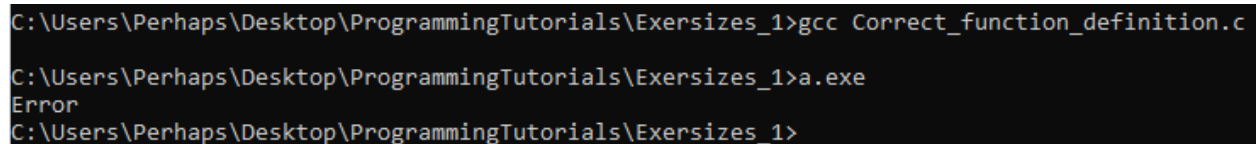
Because these are warnings and not errors, the program still manages to be run, but if more code is added, it may have unexpected behavior. Therefore, warnings should be eliminated when possible.

These warnings go away if just “hoist” our function’s definition above line 5, i.e. before the definition of main:



```
Correct_function_definition.c x
1  #include <stdio.h>
2
3  void my_print(void){
4      printf("Error");
5  }
6
7  int main(void){
8
9      my_print();
10
11     return 0;
12 }
```

No warnings!



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>gcc Correct_function_definition.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>a.exe
Error
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Exersizes_1>
```

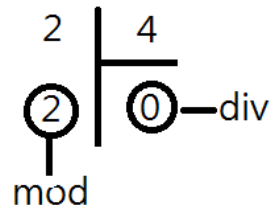
Certain scripting languages, like JavaScript, do this automatically. It is called “function hoisting” and it is very convenient, as it allows you to define your functions anywhere. But it is a good programming practice to define your functions close to where you call them (if they are defined in the same file that they are executed in). Otherwise, we’ve already seen that the `#include` statements all go to the top of the file.

## Div and mod operations

For arithmetic division, there are two operators. One that returns the result of the division (*div* operator - `</>`) and one that returns the remainder of the division (*mod*-ulo operator `<%>`).

Fun fact! In some programming languages, like python for example, the operators are the words `div` and `mod` e.g. `2 div 4` and `2 mod 4`.

For example, let's talk about 2 / 4 division. Below you can see what each operator will return:



### Floating point numbers

There is a data type that is called “double” (and another called “float”). A variable of type double (or float), stores a floating-point number (e.g., 5.6 or 3.14). The difference between the two, is the number of decimal digits that it can store (i.e., precision issues), but we won't go into further details about this topic, at least for now.

All you need to know, to solve the exercises, is that you can print a float or double value with “%f” instead of “%d” that we had for integers.

If you want more details on the subject, you can take a look at this:

<https://www.geeksforgeeks.org/difference-float-double-c-cpp/>

~~~~~