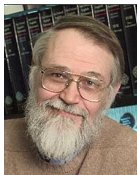
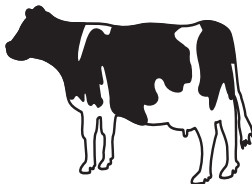


## Week 2: Introduction to Programming in C

### CS211: Programming and Operating Systems

Niall Madden

**Wednesday and Thursday, 22+23 Jan, 2020**



Brian Kernighan



Dennis Ritchie

# Reminders CS211



**Lecturer:** Dr Niall Madden, School Mathematics, Statistics and Applied Mathematics (he/him).

**Where I am:** Office: AdB-1013, Arás de Brún.

**Contact me:** Email: [Niall.Madden@NUIGalway.ie](mailto:Niall.Madden@NUIGalway.ie)

*Always include “CS211” in the subject line of your message*

**Lectures**

**Lecture** Wednesday 15:00–15:50 AC202

**Lecture** Thursday 13:00–13:50 AC204

**Lab** ~~???~~ ??? ???

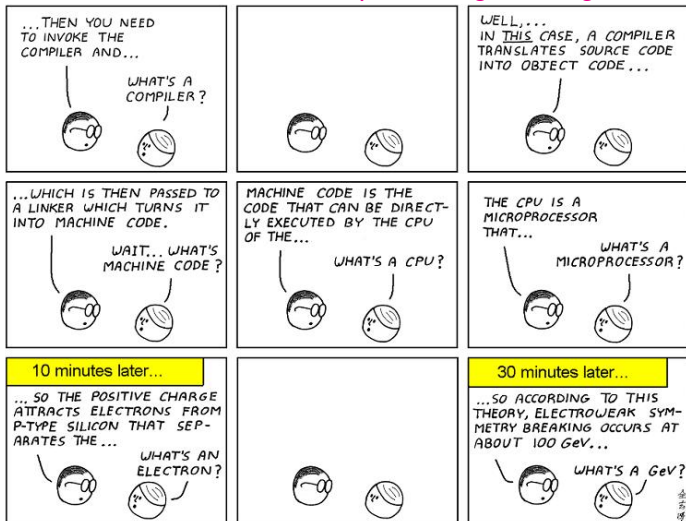
Friday 9–12 AdB-G 021

*Computer labs are a very important part of this course, and attendance is considered mandatory*

**First lab: next week.**

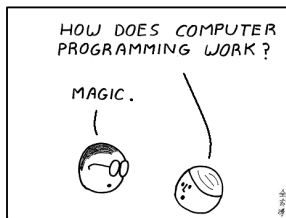
# What is a compiler?

## Abstruse Goose 98: Computer Programming 101



# This week's classes

- 1 Introduction
  - A little history
  - Books and Compilers
- 2 Course Content
- 3 Basic Structure
  - A simple example
  - A comment about comments
- 4 Variables
  - Variable names
  - Printing the value of a variable
- 5 Keywords
- 6 Operators
  - Arithmetic and Assignment
  - Relational and Logical Operators
- 7 Selection statements and loops
  - if statements
- 8 Exercises



Abstruse Goose: under the hood

The first version of the UNIX operating system (from which many/most other modern systems evolved, including MacOS and GNU/Linux) was developed at Bell Labs in 1969 for a DEC PDP-7. It was written in assembly language.

It became apparent that to develop the operating system further, it would have to be rewritten in a high level language. There wasn't one available, so they wrote one: in 1972/73 the first C compiler was written.

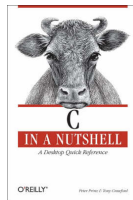
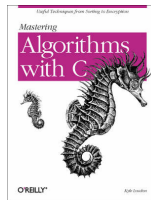
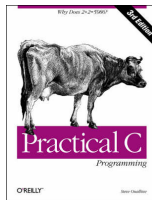
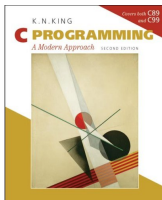
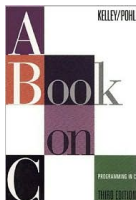
It's main goals were

- to be suitable for developing operating systems,
- be portable: compilers should be available for different computers.

The C language has continued to be developed and refined, with the most recent standard was released in 2011.

Many other languages are derived from C, or borrow heavily from its syntax, notably C++, Java, C#, PHP, Objective-C, Perl, Javascript.

**Get a good book on C.** It doesn't really matter which book! I like



These are all in the library, and “Practical C” is available online at no cost to you, through the library.

If you find some useful resource for learning C, please let me know, so that I can share with the rest of the class.



You'll also need some software to support your programming, usually:

- an IDE in which you write code;
- a **compiler** to make your code executable.

You can install these on your own computer. If doing so, I suggest **Code::Block**: <http://www.codeblocks.org/>, which is free and comprehensive (though some of the lab work we will do is OS-dependent).

Mainly, however, we will use some excellent online editors and compilers. I suggest

- [https://www.onlinegdb.com/online\\_c\\_compiler](https://www.onlinegdb.com/online_c_compiler)
- <http://cpp.sh/>
- <https://www.codechef.com/ide>
- [www.tutorialspoint.com/compile\\_c\\_online.php](http://www.tutorialspoint.com/compile_c_online.php)

# Course Content (the "Programming" bit).

In order to explore how operating systems work, you'll need to a good basic grasp of C. The basic components that we'll study are

- 1 Fundamentals of C, including
  - program structure
  - data types and variable declarations,
  - input/output,
  - arithmetic,
  - loops,
  - Flow of control (`if` statements), conditionals,
- 2 Functions.
- 3 File management and data streams.
- 4 Arrays, **pointers** and strings.
- 5 Dynamic memory management.
- 6 Abstract data types: Structures and Unions.

# Course Content

---

For the next two weeks we'll cover the fundamentals:

- |                                 |                   |
|---------------------------------|-------------------|
| (i) Basic programming structure | (ii) Variables    |
| (iii) Arithmetic                | (iv) Basic output |
| (v) <code>for</code> loops      | (vi) Basic input  |
| (vii) <code>if</code> blocks    | (viii) functions. |

# Course Content

---

Some important points about C:

- It is a **compiled** language, not an interpretive one. This means that we need a program, called the **compiler** to convert our human-readable source code into something our computer can interpret.
- It is a very **small** language; and relies heavily on external libraries that contain **functions** to achieve many important tasks, including input and output.
- But the compiler has to be told in advance how these functions should be used. So before the compilation process, the **preprocessor** is run to **include** the function descriptions that the programmer thinks are necessary.
- The code is then compiled into machine instructions (**object code**).
- The object code is **linked** with library functions to produce executable code.

01Hello.c ← link!

```
1 #include <stdio.h>
   int main(void )
3 {
   printf("Hello, World!\n");
5   return(0);
   }
```

**Line 1:** The first line begins with a # symbol. This is a “preprocessor directive”.

It directs the compiler to include a **header** file (a.k.a., “dot h file”) called **stdio.h**.

**stdio.h** is the **standard Input/Output header** file. It contains important information about the function **printf()**.

The angle brackets < and > means that the preprocessor should look in the “usual place” (varies between installations).

01Hello.c ← link!

```
1 #include <stdio.h>
2 int main(void )
3 {
4     printf("Hello, World!\n");
5     return(0);
6 }
```

**Line 2:** In C, almost everything is either

- 1 a preprocessor directive.
- 2 a variable, or variable declaration.
- 3 **a function**

The example is no different. Essentially, it is just a definition of the fundamental function `main()`. Here (and often), the function `main()` does not take any arguments, but returns an integer.

C is **case sensitive**, so `main` is different from `Main` is different from `MAIN`, etc.

*int* means "integer": it is the return value.

01Hello.c ← link!

```
1 #include <stdio.h>
2 int main(void )
3 {
4     printf("Hello, World!\n");
5     return(0);
6 }
```

*This is the argument. "Void" means 'Empty' / nothing.*

**Lines 3 and 6:** The definition of the `main()` function is encapsulated by "curly brackets": `{` and `}`

In `C` these are used to delimit various types of programme blocks.

*printf* = "formatted print"  
01Hello.c ← link!

```
1 #include <stdio.h>
2 int main(void )
3 {
4     printf("Hello, World!\n");
5     return(0);
6 }
```

**Line 4:** The function `printf()` is used to send output to `stdout`. Everything between quotes is displayed. The `\n` is a “new line”. These are not printed by default, so Line 4 above is equivalent to

```
printf("World!"); printf("\n");
```

More about *Hello* `printf` later....

Note that each (logical) line within a function is terminated by a semicolon.



01Hello.c ← link!

```
1 #include <stdio.h>
2 int main(void )
3 {
4     printf("Hello, World!\n");
5     return(0);
6 }
```

**Line 5:** The `return` keyword specifies what value should be returned to the function that called it.

Here `main` is called by the Operating System, so in this instance it specifies what the program returns to the OS on exit.

The value `0` (zero) means “everything is OK”.

Any good program should have some documentation to explain to others

- why, when and by who it was written,
- how it works.

In C, there are two types of comments:

- 1 **block comments**; Starts with `/*` and ends with `*/`.  
Everything in between, including new lines, are ignored.
- 2 **single line comments**; Everything after `//` is ignored.

It is important to add comments to you code: your future self with thank you. But, where possible, make the code self-commenting, by using sensible names for identifiers.

# Variables

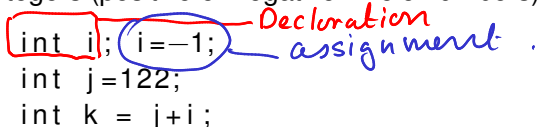
**Variables** are used to temporarily store values (numerical, text, etc, ....) and refer to them by name, rather than value.

**All variables must be defined before they can be used.** That means, we need to tell the compiler before we use them.

Every variable should have a **type**; this tells use what sort of value will be stored in it.

The variables/data types we can define include

- Integers (positive or negative whole numbers), e.g.,



```
int i; i=-1;  
int j=122;  
int k = j+i;
```

Note that one can initialize (i.e., assign a value to the variable for the first time) at the time of definition.

# Variables

- Floats – these are not whole numbers. They usually have a decimal places. E.g,

```
float pi=3.1415;
```

- Characters – single alphabetic or numeric symbols, are defined using the `char` keyword:

```
char c;      or      char s='7';
```

Note that again we can choose to initialize the character at time of definition. Also, the character should be enclosed by single quotes.

- We can declare **arrays** or **vectors** as follows:

```
int Fib[10];
```

This declares a integer array called `Fib`. To access the first element, we refer to `Fib[0]`, to access the second: `Fib[1]`, and to refer to the last entry: `Fib[9]`.

- Note that in `C`, all vectors are indexed from `0`.

In C, a variable (or function) name can be made up of up to 52 characters long and include

- *Alphabetic characters:* A, B, ..., Z, a, b, ..., z
- *Numeric characters:* 0, 1, ..., 9
- The underscore symbol: \_

However,

- it must start with a letter or underscore.
- it cannot be a keyword (e.g., `for`, `if`, `return`).

Eg `int cs211;` is ok, but not  
`int 211cs;` | Finished here  
Wed @ 3:50