Parallel Computing with GPUs

An Introduction to C
Part 1 - Hello World

The University Of Sheffield.

Dr Paul Richmond

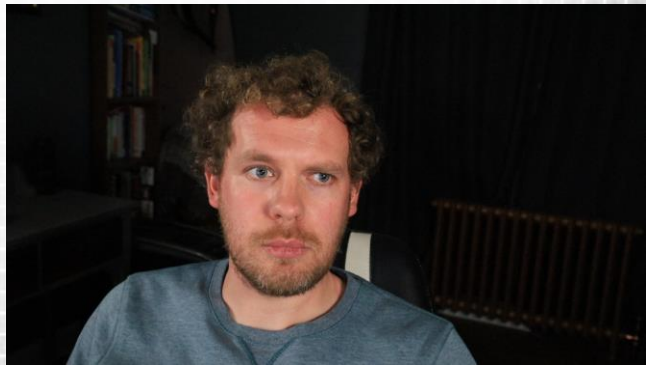
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Introduce the C programming language
  - Identify the context of the language
  - Classify compiled vs interpreted
- Basic C usage: “Hello World”
  - Recognise the basic structure of a C program
  - Categorise the different parts of the compilation process
  - Distinguish appropriate use of casting
  - Recall appropriate use of `const`

About C

- Developed in the 70s
- Low Level
  - Compiled language
  - Close to machine code (more expressive than assembly)
- Procedural Language
  - Follows **in order** a set of commands
- Weakly Typed Language
  - Some basic C data types (but no data types in assembly)
  - Unchecked casting
  - No objects, sets or strings
- Simple fundamental control flow
  - `if, else, else if`
  - `switch`
  - `do, while, for, break, continue`
  - We will ignore `GOTO:`

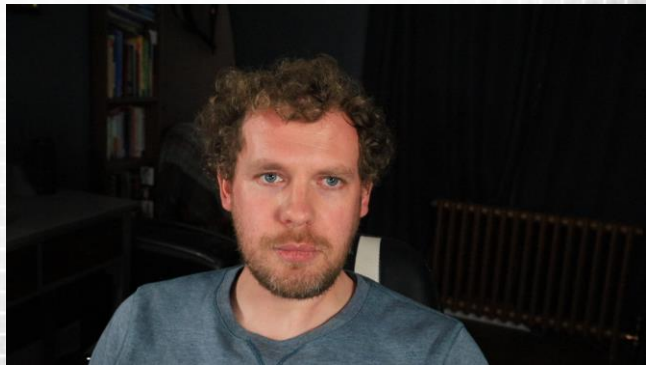


C Standardisation

- C89/ANSI C:
  - Based on famous reference manual “K&R C”
  - Proposed by American National Standards Institute
- C90:
  - ISO standard 9899:1990
  - Technically the same as C89
- C99:
  - Addition of inline, Boolean, floating point
  - Most common C standard implemented by compilers
  - ‘*strict*’ – implies the compiler follows the standard exactly
- C11:
  - Addition of multi threading support and atomics

Compiled vs Interpreted

- (C *is a*) Compiled Language
  - Compiler translates language into native machine instructions
  - Machine instructions do not port between architectures
  - Can be very powerful and high performance
- (C *is NOT an*)Interpreted Language
  - Read by an interpreter which executes the program
  - JAVA, Python etc.
  - Generally much slower (more overhead)
  - Just-in-Time (JIT): compilation at runtime to balance performance and portability



Hello World

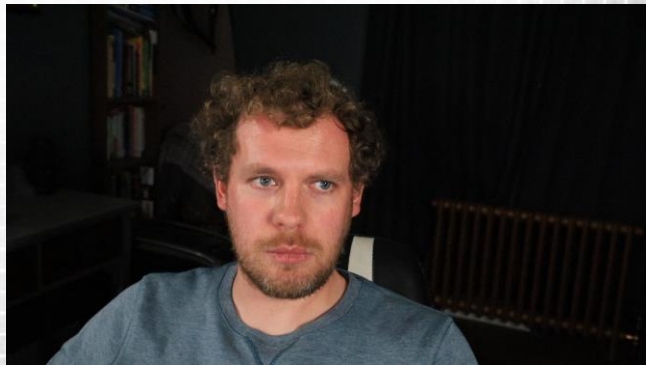
- Control flow has influenced many other languages (e.g. JAVA)
- `#include` directive: parsed by pre processor
- `printf`: basic output
- `main`: standard entry point
- Comments (`//` single line or `/* */` multiline)
- `return`: Main can return 0 to indicate success or anything else to indicate an error code

```c
/* Hello World program */

#include <stdio.h>

int main()
{
    //output some text
    printf("Hello World");
    return 0;
}
```

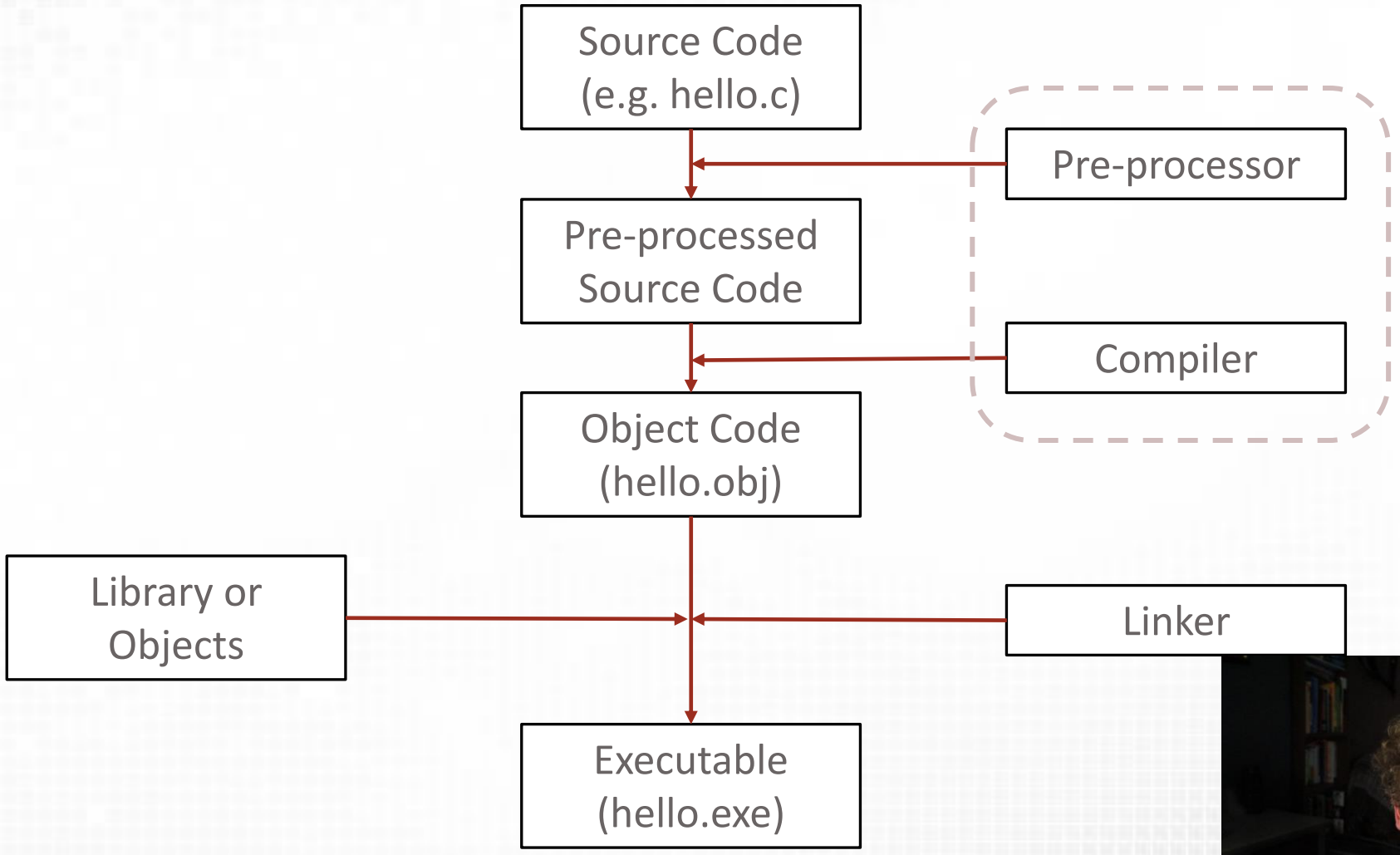
Directives and Pre-processor

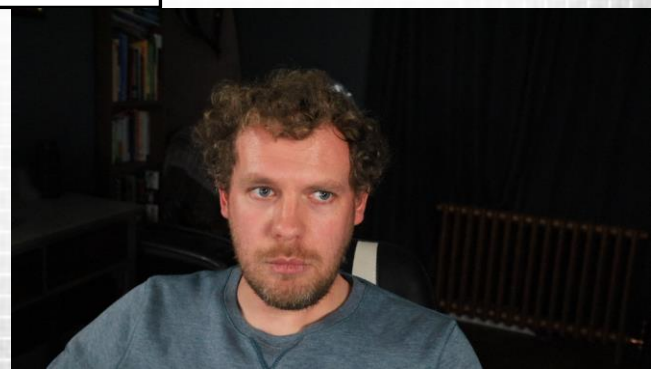
- `#include`: includes the contents of a file
  - `#include <file>`: system header files
  - `#include "file.h"`: user header files relative to working directory
- Macros
  - `#define SOME_VALUE 1024`
    - Pre-processor performs substitution in expressions.
    - E.g. `int x = SOME_VALUE;`
  - Function-like macros
    - Can have arguments
    - E.g. `#define add_one(x) (x+1)`
    - Used as: `int x = add_one(SOME_VALUE);`
  - `#if, #elseif, #else, #endif`:
    - Used to perform directive conditionals
  - `#ifdef, #ifndef`
    - If defined and if not defined: Useful for platform specific code

```
#ifdef WIN32
#include <windows_header.h>
#else
#include <linux_header.h>
#endif
```



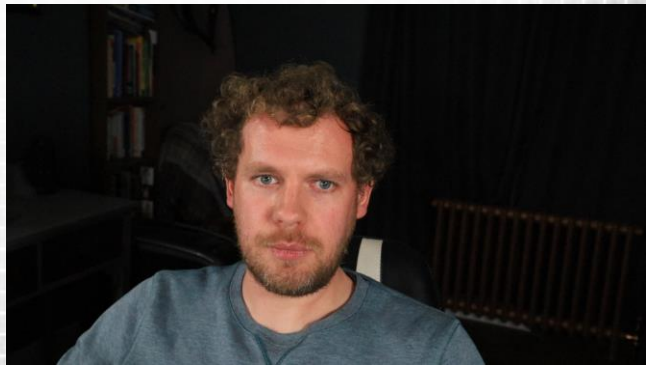
Compilation





Data types

- All sizes are compiler and machine dependant
  - `char` a single byte or single character
  - `int` a 4 byte integer
  - `float` single precision floating point (4 byte)
  - `double` double precision floating point (8 byte)
- Integer qualifiers (can omit `int`)
  - `short` `short` is 2 bytes
  - `long` `long` = `int` BUT `long long` is an 8 byte integer
- Integer and char qualifiers (affects range)
  - `signed` positive and negative
  - `unsigned` positive only
- `sizeof()` function returns size of variable or type
  - E.g. `int a; sizeof(a) = 4;`
  - `sizeof(int) = 4;`



Implicit Casting

- Implicit casting
  - When **operands** have different types the compiler will implicitly convert them
    - Also occurs in function arguments and return values
  - Implicit casting follows a promotion hierarchy (using rank)
    - `char < short < int < long < long long < float < double < long double`
    - Implicit casts always move variables up the rank
    - Order of evaluation is important!

```
int  i = 17;
char c = 'c'; // ascii value is 99
int sum;

sum = i + c;
```

Explicit Casting

- Explicit Casting
  - Cast operator `(type)` can be used on expressions or variables
  - Be careful
    - Integer truncation: `(int) 9.999999f == 9`
    - You might loose precision: `(char) 256 == 0`

```
int i, j;
double result;
i = 1;
j = 3;
result = i / j;
```

What is result?

Explicit Casting

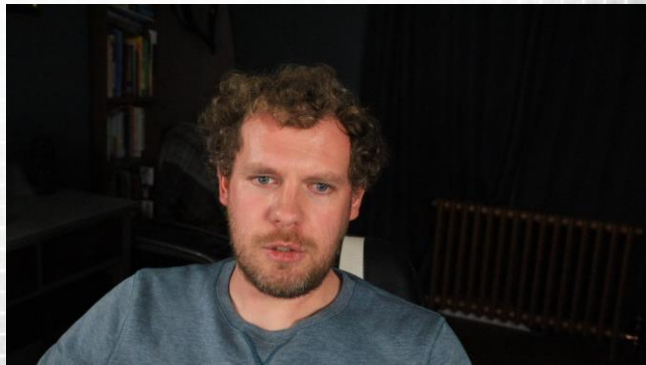
- Explicit Casting
  - Cast operator `(type)` can be used on expressions or variables
  - Be careful
    - Integer truncation: `(int) 9.999999f == 9`
    - You might loose precision: `(char) 256 == 0`

```
int i, j;
double result;
i = 1;
j = 3;
result = i / j;
```

What is result? 0

```
int i, j;
double result;
i = 1;
j = 3;
result = (double) i / j;
```

What is result?



Explicit Casting

- Explicit Casting
  - Cast operator `(type)` can be used on expressions or variables
  - Be careful
    - Integer truncation: `(int) 9.999999f == 9`
    - You might loose precision: `(char) 256 == 0`

```
int i, j;
double result;
i = 1;
j = 3;
result = i / j;
```

What is result? 0

```
int i, j;
double result;
i = 1;
j = 3;
result = (double) i / j;
```

What is result? 0.33333

const and volatile

- What does `const` mean? (e.g. `const int a = 10;`)

- What does `volatile` mean? (`e.g. volatile int a;`)

const and volatile

- What does `const` mean? (e.g. `const int a = 10;`)
  - The variable is not unintentionally modifiable
    - Compiler error if you try to modify it
  - Not quite the same as read only
    - Something else might change it if it is volatile as well!
  - Can I cast a `const` to a non `const`
    - Yes, you can intentionally modify in this way but may lead to undefined behaviour
    - Implicit casting raises a compiler error
- What does `volatile` mean? (`e.g. volatile int a;`)
  - The value may change at any time regardless of code
  - Useful in embedded systems where value may be mapped to hardware
  - Prevents compiler performing optimisations on the variable
    - Which may be unsafe if the value changes

Summary

- Introduce the C programming language
  - Identify the context of the language
  - Classify compiled vs interpreted
- Basic C usage: “Hello World”
  - Recognise the basic structure of a C program
  - Categorise the different parts of the compilation process
  - Distinguish appropriate use of casting
  - Recall appropriate use of `const`
- Next Lecture: Functions and scoping