Parallel Computing with GPUs: An Introduction to C

Dr Paul Richmond

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

This Lecture

- Introduce the C programming language
- Basic C usage “Hello World”
- Functions and scoping
- Arrays, strings and basic IO
- File IO

About C

- Developed in the 70s
- Low Level
  - Compiled language
  - Close to machine code (more expressive than assembly)
- 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

- Introduce the C programming language
- Basic C usage “Hello World”
- Functions and scoping
- Arrays, strings and basic IO
- File IO

Hello World

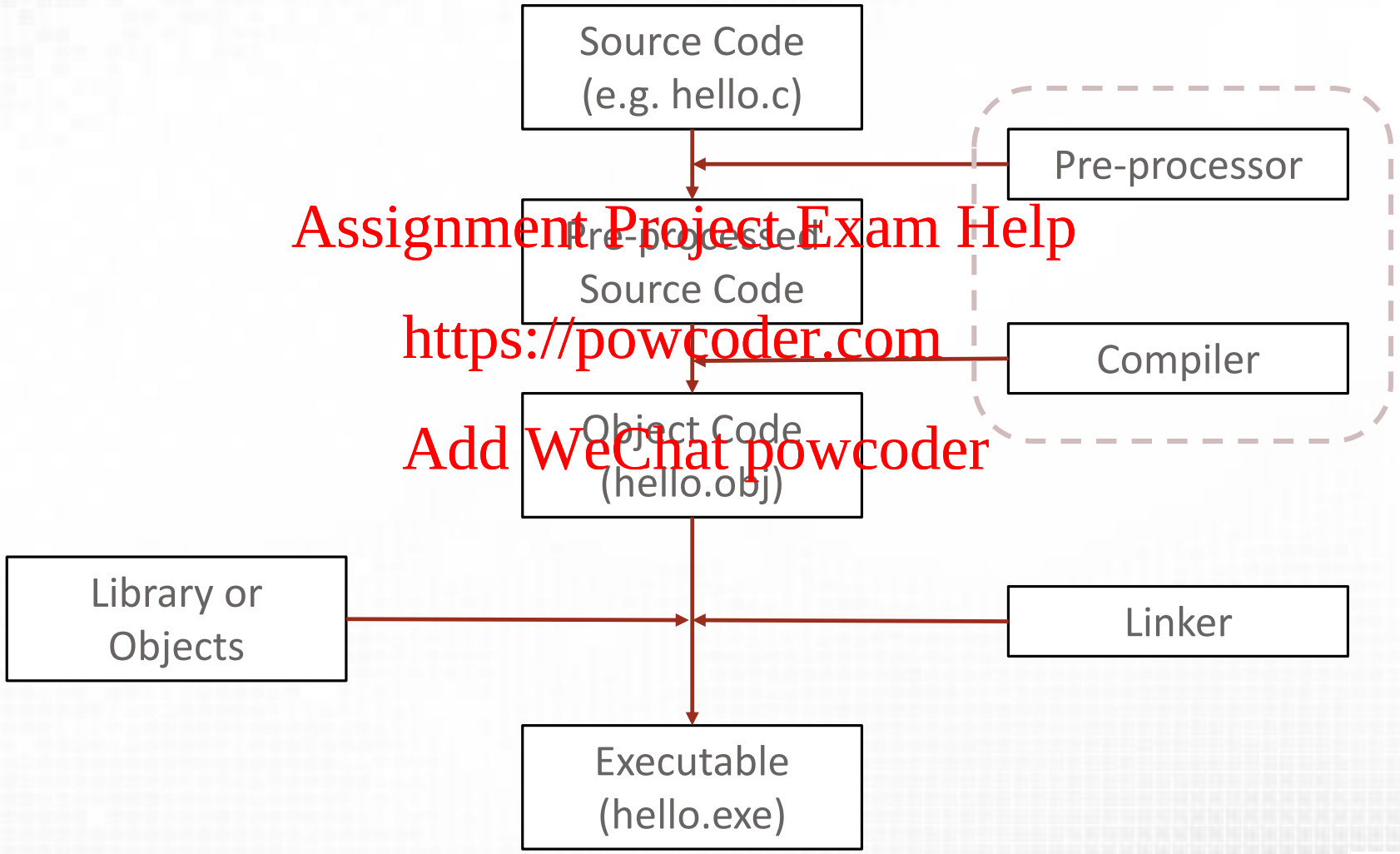
- Control flow has influenced many other languages (e.g. JAVA)
- `#include` directive: parsed by pre processor
- `printf`: basic output
- `main`: standard entry point
- Comments (`//` single line or `/* */` multiline)
- `return`: Main can return 0 to indicate success or anything else to indicate an error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


```
/* Hello World program */

#include <stdio.h>

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

Compilation



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
```

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 (4 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

- Introduce the C programming language
- Basic C usage “Hello World”
- Functions and scoping
- Arrays, strings and basic IO
- File IO

Functions

- Function definition

```
return-type function-name(optional-const argument-type argument-name, …)
{
    definitions;

    statements;

    return return-value or expression;
}
```


- Arguments are always passed by value
- No return type implies void (return can be omitted)

Scoping

```
#include <stdio.h>

int square(int a)
{
    return a*a;
}


int main()
{
    int result;
    result = square(a);

    printf("Square of 4 is %i", result);
    return 0;
}

int a = 4;
```


- Scoping lasts from where a variable or function is declared
- What is wrong with the following?

Scoping

```
#include <stdio.h>

int square(int a)
{
    return a*a;
}


int main()
{
    int result;
    result = square(a);   //ERROR

    printf("Square of 4 is %i", result);
    return 0;
}

int a = 4; //DECLARATION AND DEFINITION
```

- Scoping lasts from where a variable or function is **declared**

- What is wrong with the following?

error C2065: 'a' : undeclared identifier

Function Scoping

```
/* Hello World program */

#include <stdio.h>

int main()
{
    int result, a;
    a = 4;
    result = square(a); //ERROR

    printf("Square of 4 is %i", result);
    return 0;
}

int square(int a)
{
    return a*a;
}
```

- Another example with a function

error C2065: 'square' : undeclared identifier

Function Scoping

```
/* Hello World program */

#include <stdio.h>

int square(int a)
{
    return a*a;
}

int main()
{
    int result, a;
    a = 4;
    result = square(a);

    printf("Square of 4 is %i", result);
    return 0;
}
```


This works but not always practical

Function Declarations

```
/* Hello World program */

#include <stdio.h>

int square(int);

int main()
{
    int result, a;
    a = 4;
    result = square(a);

    printf("Square of 4 is %i", result);
    return 0;
}

int square(int a)
{
    return a*a;
}
```

- *A function declaration can be used to forward declare functions*
  - *Sometimes Referred to as a prototype*
  - *Argument names not necessary*

A declaration is different to the definition

Variable Declarations

```
#include <stdio.h>

int square(int);//function declaration
extern int a;   //DECLARATION

int main()
{
    int result;
    result = square(a);

    printf("Square of 4 is %i", result);
    return 0;
}

int a = 4; //DEFINITION

int square(int a)
{
    return a*a;
}
```


- Declarations are not just for functions.
- **`extern`** can be used to declare a variable or function
  - That is defined **elsewhere**
  - **BUT** only once

extern

main.c

```c
#include <stdio.h>

//DECLARATIONS
extern int square(int);
extern int a;

int main()
{
    int result;
    result = square(a);

    printf("Square of 4 is %i", result);
    return 0;
}
```

my_maths.c

```c
//DEFINITIONS

int a = 4;

int square(int a)
{
    return a*a;
}
```


- `extern` can declare variables and functions defined in other source modules
  - Resolved by linker

headers

my_maths.h

```
//DECLARATIONS
extern int square(int);
extern int a;
```

my_maths.c

```
//DEFINITIONS
#include "my_maths.h"

int a = 4;

int square(int a)
{
    return a*a;
}
```

main.c

```
#include <stdio.h>
//include
#include "my_maths.h"

int main()
{
    int result;
    result = square(a);
    printf("Square of 4 is %i", result);
    return 0;
}
```

other.c

```
//include
#include "my_maths.h"

int add_a_b_squares(int b)
{
    return square(a) + square(b);
}
```


- Headers can be used to share common declarations

Declaration vs Defintion

- *A **declaration** introduces an identifier and describes its type, be it a type, or function. A **declaration** is what the compiler needs to accept references to that identifier. E.g. at global scope*


```
extern int a;
void sum(int a, int b);
extern void sum(int a, int b);
```


- A ***definition*** actually instantiates/implements this identifier. It's ***what the linker needs*** in order to link references to those entities. These are definitions corresponding to the above declarations:

```
int a;
int a = 1;
int sum(int a, int b){ return a + b; }
extern void sum(int a, int b) { return a + b; }
```

More Examples: https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/

Static

- What is a `static` variable?

Static

- What is a `static` variable?
  - A static **global** variable or function is visible only in the compilation unit it is declared
    - i.e. No use of `extern` in other source modules
  - A static **local** variable (inside a function) keeps its values between invocations
    - It is defined only once but is declared for lifetime of program


```
void static_test()
{
  int a = 10;
  static int b = 10;
  a += 5;
  b += 5;
  printf("a = %d, sa = %d\n", a, b);
}


int main()
{
  int i;
  for (i = 0; i < 5; ++i)
    static_test();
}
```

```
a = 15, b = ??
a = 15, b = ??
a = 15, b = ??
a = 15, b = ??
a = 15, b = ??
```

What are the values of `b`?

Static

- What is a `static` variable?
  - A static **global** variable or function is visible only in the compilation unit it is declared
    - i.e. No use of `extern` in other source modules
  - A static **local** variable (inside a function) keeps its values between invocations
    - It is defined only once but is declared for lifetime of program


```
void static_test()
{
  int a = 10;
  static int b = 10;
  a += 5;
  b += 5;
  printf("a = %d, sa = %d\n", a, b);
}


int main()
{
  int i;
  for (i = 0; i < 5; ++i)
    static_test();
}
```

```
a = 15, b = 15
a = 15, b = 20
a = 15, b = 25
a = 15, b = 30
a = 15, b = 35
```

What are the values of `b`?

- Introduce the C programming language
- Basic C usage “Hello World”
- Functions and scoping
- Arrays, strings and basic IO
- File IO

Arrays

- Arrays can be compile time defined using `[size]`
  - Local arrays will be created on the stack (not heap)
  - Multidimensional Arrays possible
- Character Arrays
  - Represent strings
  - String literals can be assigned to an array at declaration only
  - Termination required with `'\0'` character
  - `char *name` is equivalent to `char name[]`


```
char my_string1[] = "hello";
char my_string2[6] = "hello";
char my_string3[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char *mystring4 = "hello";

char my_string5[6];
my_string5 = "hello"; //ERROR

char my_string6[5] = "hello" //ERROR
```

Heap vs. Stack

- Stack
  - Memory is managed for you
  - When a function declares a variable it is pushed onto the stack
  - When a function exists all variables on the stack are popped
  - Stack variables are therefore local
  - The stack has size limits (1Mb in VS2017)
- Heap (next lecture)
  - You must manage memory
  - No size restrictions (except available memory)
  - Accessible by any function
- Other
  - Global variables stored in a special data area of memory
  - Program stored in code area of memory

Extra Reading: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Basic IO

- Text Stream abstraction for all input output
  - `stdin`: Standard input
  - `stdout`: Standard output
  - `stderr`: Standard Error
  - `stdin` and `stdout` can be manipulated by;
    - `int getchar();`
    - `int putchar(int c);`


```
#include <stdio.h>
#include <ctype.h>

void main()
{
    int c;
    while ((c = getchar()) != '\n')
        putchar(toupper(c));
}
```

Formatted IO

- Output: `printf`
  - Print using formatted string
  - Format specification string and variables
- Input: `scanf`
  - Scans input according to format string
  - Saves input to variables in given format
  - Return value is the number of arguments filled
  - Variable argument are pointer to variables (`&`)
    - More on this next lecture…


```
printf("integer variable a value is %d", a);
printf("float variable b value is %f", b);
scanf("%d", &myint);
scanf("%f", &myfloat);
```

String formatting: Common format specifiers

```
%[flags][width][.precision][length]specifier
```

Specifier	Output	Example
d or I (lld)	Signed integer (long long signed integer)	123, -123
U (llu)	Unsigned integer (long long unsigned integer)	123
x or X	Unsigned hexadecimal integer (X uppercase)	1c4, 1C4
f	Decimal floating point	123.456
e or E	Scientific notation (E uppercase)	6.64e+2, 6.64E+2
c	character	A
s	Terminated string of characters	character string

Flag	Description
–	Left justify given width
+	Forces use of + or - sign
0	Left pads the number with zeros (0)

.precision	Description
.number	For d, u or i to minimum number of digits For f and e the number of decimal places after decimal point

String Formatting Escape Characters

Escape Sequence	Character represented
\a	Alarm beep (system beep)
\b	Backspace
\f	Formfeed (new page), e.g. new page in terminal
\n	New line or line feed
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\’ or \” or \?	Single or double quotes or question mark

Formatting examples

```
printf("\t%0.4d\n", 1);

printf("\t%0.4d\n", 12345678);

printf("%d\n", (int)1.23456);

printf("%d\n", sizeof(1.95f));
```


What will each of the following output?

Formatting examples

```
printf("\t%0.4d\n", 1);

printf("\t%0.4d\n", 12345678);

printf("%d\n", (int)1.23456);

printf("%d\n", sizeof(1.95f));
```



```
        0001
        12345678
1
4
```


What will each of the following output?

Formatted string input and output

- `sprintf`
  - The same as `printf` but operates on a character array
- `sscanf`
  - The same as `scanf` but operates on a character array


```
char s1[] = "COM4521";
int module;
char buffer[32];

sscanf(s1, "COM%d", &module);
sprintf(buffer, "COM%d is awesome!", module);
```

IO example

- A basic calculator for summing inputs

```
#include <stdio.h>

int main()
{
    int a, sum;
    sum = 0;

    while (scanf("%d", &a) == 1)
        printf ("\tsum:%0.8d\n", sum += a);

    return 0;
}
```

- Introduce the C programming language
- Basic C usage “Hello World”
- Functions and scoping
- Arrays, strings and basic IO
- File IO

Files

- Files are still a stream
  - `FILE* fopen(char *name, char *mode);`
    - Mode: “r” = read, “w” = write, “a” = append, “b” = binary, “+” = open for update
  - `int fclose(FILE *file);`


```c
#include <stdio.h>
#include <string.h>

void main()
{
    FILE *f = NULL;
    f = fopen("myfile.txt", "r");

    if (f == NULL){
        fprintf(stderr, "Could not open file\n");
    } else {
        fclose(f);
    }
}
```

File reading and writing of strings

- By character
  - `int getc(FILE *file);` same as `getchar` but on a file stream
  - `int putc(int c, FILE * file);` same as `putchar` but on file stream
- By formatted lines
  - `int fscanf(FILE *f, char *format, …);`
  - `int fprintf(FILE *f, char *format, …);`


```
void filecopy(FILE* f1, FILE *f2)
{
    int c;
    while (c = getc(f1) != EOF)
        putc(c, f2);
}
```

Character array operations

```
char str1[20];
char str2[20];
strcpy(str1, "To be ");
strcpy(str2, "or not to be");
strncat(str1, str2, 6);
```

- `#include <string.h>`
- Copying
  - `char * strcpy ( char * destination, const char * source );`
- Compare
  - `int strcmp ( const char * str1, const char * str2 );`
  - Returns 0 if equal
- Concatenate
  - `char * strcat ( char * destination, const char * source );`
- Length
  - `size_t strlen ( const char * str );`
  - `size_t` is an unsigned integer of at least 16 bits
- `n` versions
  - Each function has a version which performs the operation up to `num` characters
  - E.g. `strncpy`, `strncmp`, `strncat` all take an extra argument (…`size_t num`)

String Coversions

- `#include <stdlib>`
- `atof`: convert to float
- `atoi`: convert to int
- `strtod`: convert to double
- `strtoul`: convert to unsigned long


```
char *x = "450";

int result = atoi(x);

printf("integer value of the string is %d\n", result);
```

Summary

- C is a low level, weakly typed and compiled language
- Global variables and functions can be made available in other modules by declaring them as external (`extern`)
  - Header files can be used to group common declarations
  - A declaration is not the same as a definition
- C has basic character and line operators for streams
  - The console and files are both examples of streams
- There is no string datatype only arrays of characters
  - There are some string manipulation functions which operate on char arrays
- We hinted at the `*` operator
  - `char name[] == char *name`
  - This is the topic of the next lecture