Systems Programming — Lecture 1: Introduction to C Programming

Dr Konrad Dabrowski
konrad.dabrowski@durham.ac.uk

E103 Christopherson Building

1 Key topics for sub-module

- Syntax and semantics of the C programming language
- Memory access and management
- Design of large programs in non-object-oriented language
- Unix/Linux shell programming
- A dash of C++

2 Course details

- Intro, HelloWorld, Compiling, Pre-processor
- Control flow and functions
- Data types, structs and unions
- Memory access using pointers
- Dynamic memory management
- Scope of variables and recursive functions
- Large programs and external libraries
- Debugging
- UNIX/Linux and C
- C++

First practical: Week 3
Summative Assessment: Coursework (hand-out 7th February, hand-in 29th February)

3 Resources and Books

- The traditional text for C programming is *"The C Programming Language"*, Kernighan and Ritchie, Second Edition, Prentice Hall, ISBN 0-13-110362-8 (good reference book, although some aspects are a little dated. Exercise answers: `https://web.archive.org/web/*/http://www.trunix.org/programlama/c/kandr2/`)
- Based on the Kernighan and Ritchie book Steve Summit has a good set of free tutorial notes on C programming: `http://www.eskimo.com/~scs/cclass/`
- An excellent and comprehensive modern book is: *"C Programming A Modern Approach"*, K.N. King, Second Edition, ISBN 978-0-393-97950-3
- See `https://stackoverflow.com/questions/562303/the-definitive-c-book-guide-and-list` for other book suggestions.

4 Course Requirements

- Some background assumed in programming
- There are some references and comparisons to Java

5 Standardisation of the C Language

K&R C Described in Kernighan and Ritchie, The C Programming Language (1978) De facto standard

C89/C90 ANSI standard X3.159-1989 (completed in 1988; formally approved in December 1989) International standard ISO/IEC 9899:1990

C99 International standard ISO/IEC 9899:1999 Incorporates changes from Amendment 1 (1995)

C11 International standard ISO/IEC 9899:2011

C18 ISO/IEC 9899:2018 – the current standard for C

C2x Next version of the standard expected in 2021/2022

6 C-based Languages

C++ includes (almost) all the features of C, but adds classes and other features to support object-oriented programming

Java is based on C++ and therefore inherits many C features

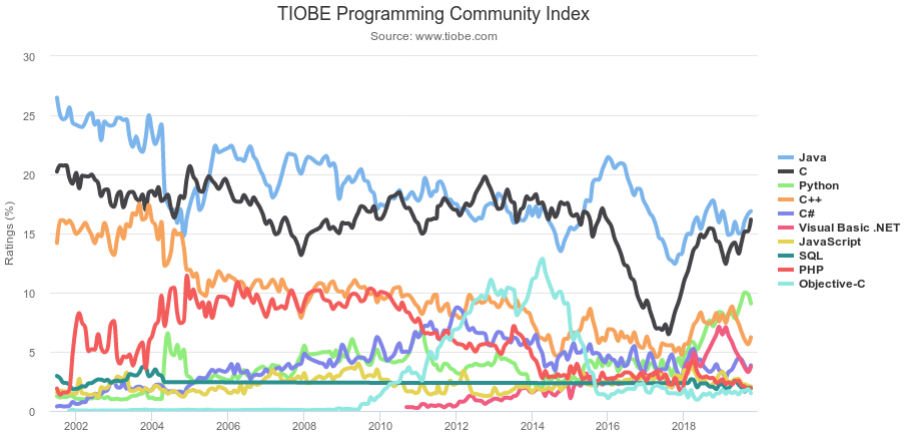
Objective-C is another OO extension for C, used in OSX/iOS development

C# is a more recent language derived from C++ and Java

Perl has adopted many of the features of C

+ many others

7 Why Choose C?



8 The C Language

- Low-level – close to assembly language
- Small core language
- Lots of well tested and freely available libraries

- Compiled not interpreted (in principle)
- Permissive – dangerous – no warnings

9 A First Program

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

Saved in a file with a "`.c`" file extension, for example "`helloworld.c`"

9.1 Preprocessor directives

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

- Lines that start with a `#` are commands to the C pre-processor

  `#include<stdio.h>`

- looks for the source code file `stdio.h` and includes it before compilation
- `stdio.h` is a file required to use the standard input and output library

9.2 The `main()` Function Declaration

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

- All C programs have an entry function called `main()`. This is called by the runtime system to start your program running.

9.3 The `printf()` Function Call

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

- Function call to `printf()` which implements formatted text printing to the console window.
- The string argument includes an escape sequence '`\n`'
  - this generates a newline character

9.4 Function `return` Statement

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

- UNIX programs often return a zero value to indicate they have exited normally
- If there is no return statement, this will not cause a problem at compile-time
- If the return value is of the wrong type this may cause a warning at compile-time or a problem at run-time

10 A Second Program

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

- This produces identical output to the first program

11 A Temperature Converter

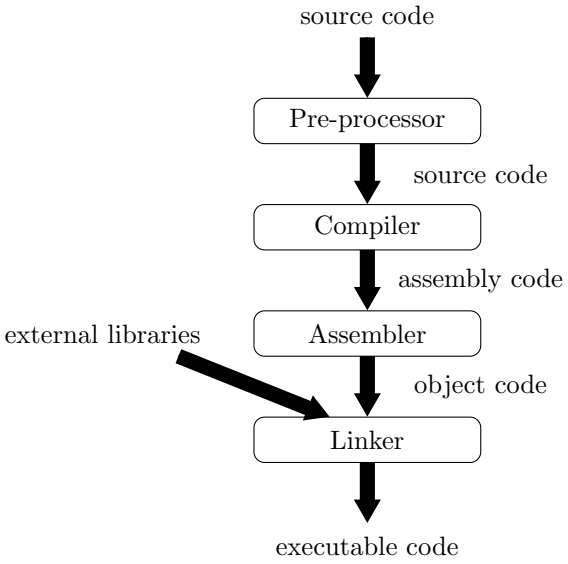
```
#include <stdio.h>
int main()
{
        int F = 10;
        int C;
        C = ((F - 32) * 5) / 9;
        printf(" %d F = %d C \n", F, C );
        return 0;
}
```

- C will truncate when encountering a non integer to be converted to integer
- This code fragment converts a temperature from Fahrenheit to Celsius and prints the result
- We could change C to a `double`
  - Store a floating point number
  - We would need to change the output format

12 `printf()`

- So popular it was added to Java in 5.0
- Variable number of parameters (also added to Java 5.0)
- First parameter explains how the rest are to be formatted using
  - `%d` signed decimal (`int`)
  - `%u` unsigned decimal
  - `%o` octal
  - `%x` hexadecimal
  - `%f` floating point so `%4.2f` will give `3.14`
  - `%e` floating point (exponent form)
  - `%c` character
  - `%s` string
- Number after is the number of characters to output
- Dot followed by number – number of decimal places

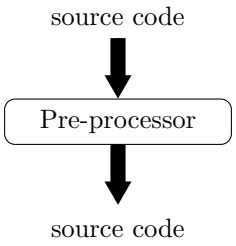
13 Compilation Model



14 `gcc` Options

- When compiling on Linux, use `gcc`
- Use `-E` option to do pre-processing only, or call `cpp`
- Use `-S` option to go as far as compilation only
- Use `-c` option to go as far as assembly only
- Use `nm` tool to investigate object libraries
- Common libraries (`.o` and `.a`) stored in `/usr/lib`
- Use `ld` linker separately

15 The C Preprocessor



- Directives such as `#define` and `#include` are handled by the *pre-processor*, a piece of software that edits C programs just prior to compilation
- Its reliance on a pre-processor makes C (and C++) unique among major programming languages

16 The C Pre-processor `#include`

- For system header files use:

  `#include<stdio.h>`

- Looks for the file `stdio.h` in C's include file directories
- On UNIX by convention this is `/usr/include`
- For user header files use:

  `#include"fibonacci.h"`

- Searches in current directory first then in system directories

  `-I path`

- Adds the directory `path` to the search path for include files when using `gcc`

17 Definitions

- Used to provide definitions in code (takes up no memory as it is just text replace):

```
#define A_NAME A_VALUE

#define MY_AGE 18

...

int nextBirthday = MY_AGE + 1;
```

- Can also specify name and value at compile time:

```
gcc -DMY_AGE=18 myProgram.c
```

- Pre-processor performs a search and replace of `A_NAME` for `A_VALUE`

18 Conditionals

These will remove irrelevant bits of code before compilation

```
#ifdef A_NAME // tests if A_NAME is #defined
        <program text>
#else
        <program text>
#endif
```

- Can also test for the lack of `A_NAME`:

```
#ifndef A_NAME // tests if A_NAME is not #defined
        <program text>
#else
        <program text>
#endif
```

19 Conditional compilation for debugging

```
#define MY_DEBUG // define an identifier

#ifdef MY_DEBUG
        assert( i > 0 );
        printf( "i is  %d  \n",  i );
#endif
```

- This allows the inclusion of your debugging code only when `MY_DEBUG` is defined
- No overhead is generated when it is not defined since no code is included for compilation (compared to a standard `if` statement)
- Can also use `#ifndef` tests if an identifier is not defined

20 Parameterized macro definitions

- Definition of a *parameterized macro* (also known as a *function-like macro*):

`#define` *identifier*`(` x_1 `,` x_2 `,` $\ldots$ `,` x_n `)` *replacement-list*

- x_1 , x_2 , $\ldots$, x_n are the macro's parameters
- e.g. `#define ADD(a,b) a+b`
- The parameters may appear as many times as desired in the replacement list
- N.B. There must be no space between the macro name and the left parenthesis
- If space is left, the preprocessor will treat `(`x_1 , x_2 , $\ldots$, x_n`)` as part of the replacement list

21 Parameterised macro definitions

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

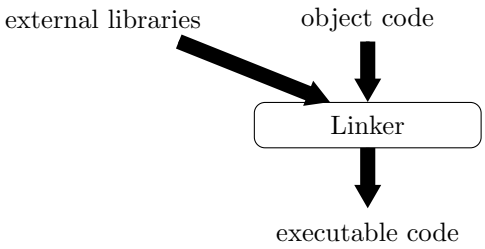
22 Parameterised macro definitions

- Using a parameterized macro instead of a true function has a couple of advantages:
  - The program may be slightly faster. A function call usually requires some overhead during program execution, but a macro invocation does not.
  - Macros are "generic." A macro can accept arguments of any type, provided that the resulting program is valid.

23 Parameterised macro definitions

- Potential disadvantages:
  - *Arguments aren't type-checked:* When a C function is called, the compiler checks each argument to see if it has the appropriate type. Macro arguments aren't checked by the preprocessor, nor are they converted
  - They work as direct substitutions in your code. *Always use brackets to fullest extent possible*
    - e.g. `#define DOUBLE(x) 2*x` might not do what you expect. Why not?

24 The link editor (linker)



- The linker's job is to combine all the files needed to form the executable
- It specifically has to resolve all symbols, functions and variables, it most often fails when it can't find required object code, for example because it is in the wrong folder

25 Next time

- Control flow and functions