Introduction to C

FACULTY OF
ENGINEERING

Dr. John Stavrakakis
COMP2017/COMP9017

THE UNIVERSITY OF
SYDNEY

Acknowledgement

Some material in these slides was based on
lectures by Prof Bernhard Scholz, A/Prof Bob
Kummerfeld and Prof Judy Kay

› Initially UNIX was written in low-level PDP-7 assembly

› Ken Thompson invented B based on BCPL to overcome issues of PDP-7 assembly.

› Dennis Ritchie built on B and called his language C.

› Unix on PDP-11 was rewritten in C

- First Unix kernel in C in the year 1973

› Kernighan and Ritchie published book in 1978

- "The C Programming Language"

› Later C was standardized

- C89 standard (or ANSI-C) introduced better parameter handling and library standards.

- C90 standard few minor modifications

- C99 latest standard (we use this for this class)

- C1X -> new standard will come up soon

› C-compilers employ two languages:

- Preprocessing Language: text-macro language

- Actual C-language: high-level programming language

PDP-11

› Mainly used for

- Systems software (OS, embedded systems, etc.)

- Software that needs hardware interaction

› Also..

- Application programming, science/engineering, etc.

› C-Compilers exist for nearly all computer architectures

› A very popular language

› C does not have features such as

- Objects and Classes

- Templates

- Operator/Function overloading

› C++ overcomes this and is a successor of C

› Writing a non-optimizing C-Compiler is straightforward

- Reason for the success story of C
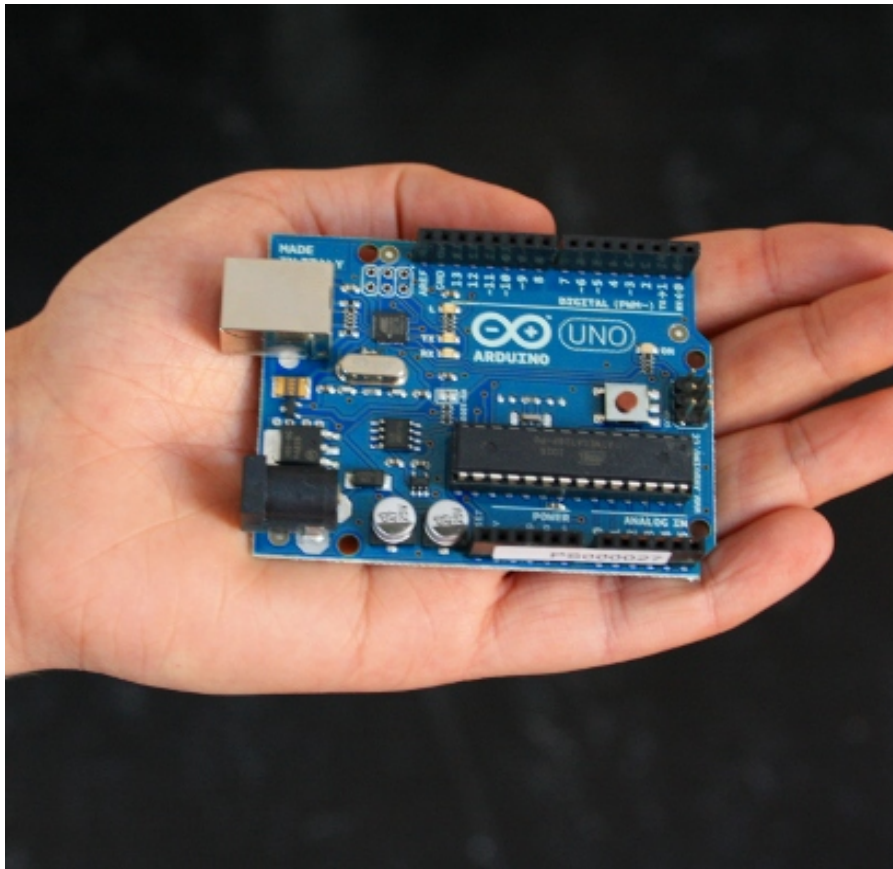
› Arduino Platform

› Project started in 2005

› Open-source electronics prototyping platform

› IDE: Arduino development environment

› ATMEL processor

› Cheap to purchase: $35

› Programmed in C (C++)

- Better than assembly

- Full control of μC

- No operating system

› More than 1M devices sold!

› Started in 1991 by Linux Torvalds

› Linus' UNIX -> Linux

› Kernel, i.e., core of the operating system.

› To complete distribution, GNU tools were used.

› In 1992 the first distributions emerged.

› Now we have numerous devices running Linux

- Smartphones, routers, …

› C is the language of choice

- HW Indepdence & Performance

› Python is a scripting language.

› Was released in 2000; has been spreading rapidly because ease of use.

› Comprises several programming paradigms

  - Imperative

  - Object-oriented

  - Functional

› Easy to learn

› Standard reference implementation is written in C.

Guido van Rossum

› Apache Web-server is back-bone of the internet

› Initially released 1995

› surpass the 100 million website milestone in 2009

› Runs widely on Windows, Unix, Mac, …

› Written in C

› C-Programs consists of two language components

- Preprocessing Language

- C-Language

› Preprocessing Language

- Text-macro language

- Definition of macros

- Include files

- Conditional compilation

› Differences

- Control flow structures are the same

  - esp. before Java 1.4

- References are called "pointers" in C

- No garbage collection

  - Programmer is responsible for allocating and freeing memory

- No classes or objects

› A C-program consists of a set of files containing:

- global variables

- function definitions

  - "main" is the first function invoked

- functions have local variables

```java
/* This program prints "Hello world." on a line and exits
 */
public class HelloWorld
{
  public static void main (String args[])
  {
        System.out.println ("Hello world.");
  }
}
```

```c
#include <stdio.h>

int main (int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

› Prints "Hello world!" on standard output

› Does not read from standard input

› Variable argc stores number of arguments

› Variable argv pointers to arguments

```c
int main(int argc, char **argv)
{
  int     ftemp;  /* the fahrenheit temperature */

  printf("Please enter a fahrenheit temperature");
  scanf("%d", &ftemp);
  printf("%d fahrenheit is %d centigrade", ftemp,
                          (ftemp - 32) * 5 /  9);

  return 0;
}
```

› Create a program text in a file whose name has the suffix ".c"

› Compile the program using the command gcc

- gcc hello.c –o hello

- gcc hello.c

› Use all those compiler flags

› Run the program by typing the name of the object file produced by the compiler. (The default is a.out.)

› ./hello

› ./a.out

› C closer to underlying machine

› C has simple memory model

- pointers, bit-level operators

- arrays very close to memory model

› C assume programmer knows best

› Java object-oriented v C procedural

- No object

- No polymorphism

- No inheritance

› Block structured

› Most control structures

- if, else, while, do ... while, switch, for (mostly),

- break, continue (no labels in C)

› Arrays

› Operators (mainly the same)

› Basic data types (mainly similar)

> C macros (#define)

> Call-by-name

> C has declaration for variables and functions, often in header files that are included

> conditional compilation

› Arrays can be handled with pointers

› Arrays can be created and initialised in declaration

› C strings are just arrays (with termination character)

› sizeof operator

› create dynamic data structures with malloc()

› C allows declarations only at block start

› C and Java have some similarities

- syntax, control structures

› but some deep differences

- Java is OO, C much closer to the hardware

› C is higher performance than Java

› C is widely used for embedded systems, operating systems etc

› C has evolved into OO forms (Objective C, C++)

A function consists of

- A function declaration:

  - Name of function,

  - Return type of function,

  - Parameter list and their types

```
int foo(float f1, char c2)
…..
```

- Followed by a function body:

  - Local variables & control flow

```
int foo(float f1, char c2)
{
    int x = 0;
    …
    return x;
}
```

> External or forward function declarations do not have a function body, just a semicolon

  - parameter types are specified without variable names

    `int foo(float, char);`   `extern int foo(float, char);`

> A function with a given name can only be defined once

> If no return value exists for a function, use the type **void**

    `void foo(….) { ….}`

> If no parameters exist use, use type **void**

    `void foo(void) { ….}`

› Functions with arbitrary numbers of parameters are possible

```
int printf(const char *format, …)
```

› In this case, a special interface is required for querying values of parameters

- Lookup the **va_args** interface

- At least one fixed parameter in the function is necessary

- Function call is simple

```
printf("%d, %f", 10, 10.5);
```

› Compute factorial n!

```
int factorial (int n)          ← function declaration
{
        int result;            ← local variable

        if (n > 1 )            ← control flow
        {
            result = n * factorial(n-1);
        }
        else                   ← control flow
        {

            result = 1;

        }

        return result;

}
```

› Mostly the same as in Java

› statements are terminated by a semicolon; the null statement is allowed.

<stmnt>;….; <stmnt>;

› A statement can be a sequence of statements inside a **block**

```
{
    <stmnt>;
    <stmnt>;
}
```

› if statements:

> if (<expr>)
>   <stmnt>

> if (<expr>)
>   <stmnt>
> else
>   <stmnt>

› while statements:

> while (<expr>)
>   <stmnt>

> do <stmnt>
> while (<expr>)

› for statement:

```
for ( <initial-expr>; <boolean-expr>; <continuation-expr> )
   <stmnt>
```

› for statement example:

```
for ( x = 0; x < 100; x++)
    counter[x] = x;
```

› return, break and continue statements:

```
return <optional expression>;

break;

continue;
```

› **return** will return to the calling function, optionally returning a value.

› **break** will jump out of the smallest enclosing loop or switch

› **continue** will jump to the next iteration of the smallest enclosing loop

› switch statement:

```
switch(…)
{
case <const-expr>: <statement-sequence>;
case …: …
default: …
}
```

› Programs consist of "modules"

 - **A module is a file**, i.e., hello.c

› Modules consist of

 - Function declarations

 - Function definitions

 - Global variables

› Modules are translated to object files

› Object files are linked by linker with other object files and standard libraries

THE UNIVERSITY OF SYDNEY

› A module can refer to global variables and functions of other modules

- use the **extern** qualifier for global variables

› Symbols can only be *defined* in one module

› Data structures definitions and declarations, macro definitions and external function declarations are found in modules

- These are commonly found in header files

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

int global1;

int foo(int x,int y)
{
    return x + y;
}
```

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

extern int global1;

extern int foo(int x,int y);

int foo2(int x,int y)
{
    return foo(x,y)+global1;
}
```

foo.c

```
int foo()
{
        printf ("hello from foo\n");
        return 0;
}
```

foo.h

```
extern int foo();
```

foo.c

```
int foo()
{
      printf ("hello from foo\n");
      return 0;
}
```

foo.h

```
extern int foo();
```

sample.c

```
#include "foo.h"

int main(int argc, char **argv)
{
      foo();
      return 0;
}
```

› Basic Input: int getchar(void);

   - reads from standard input next character

   - returns -1 (defined as the symbol EOF) if end of input reached

› Basic Output: void putchar(int c);

   - Write a character (represented as an integer) to standard output

› getchar/putchar are very simple

› printf()-function writes to standard output:

- Strings

- variables of primitive a data-type

```
int printf(const char *format, …);
```

› Return value:

- Number of printed characters

› Arguments

- First argument is a format string

- Followed my an arbitrary number of parameters depending on format string

› Example:

printf("%d %f\n", 10, 10.5);

- Output: 10 10.5

- %d print an integer followed later as a parameter

- %f print a float followed later as a parameter

- \n means print new line

Code	Description
%c	Character
%d	Integer
%u	Unsigned integer
%f, %g, %e	Double floating point number
%x	Hexadecimal
%ld	long
%.2f	Print floating point numbers with two decimal points
%s	String
%p	Pointer
%%	Print %

› scanf()-function reads from standard input:

- Values of primitive data-type and strings

› Return value:

```
int scanf(const char *format, …);
```

- Number of successfully read items

› Argument

- First argument is a format string

- Followed my an arbitrary number of parameters depending on format string

- *Parameters must be pointers* – not values

› Example:

- Read an integer and store it in x

- Read a float and store it in f

- Same format string as in scanf

```
int x;
float f;
scanf("%d %f", &x, &f);
```

End of Section