University of Nottingham
UK | CHINA | MALAYSIA

Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

Chapter 1

Introduction

Welcome to MMME3085 Computer Engineering and Mechatronics!

- Course tutors:
- Course convener + Mechatronics: Abdelkhalick Mohammad
- Computer Engineering: Louise Brown
- Lab Sessions: Surojit Sen

- Computer Engineering lectures: Chemistry C15 Monday 1-3pm
- Mechatronics lectures: Psychology A1 Thursday 9-11am
- Computer labs: Coates C19 Tuesday 11am – 1pm
- Labs: AMB C09/10 Wednesday and Friday (see timetable for dates and times)

Record

Outline of the Module (1)

	Week		Assessment		Programming		Mechatronics					
w/c ↓	University	Teaching			Lecture	Lab	Lecture	Seminar	Lab-1	Lab-2	Lab-5	Lab-6
			Room →		*Chemistry C15*	*Coates C19*	*Psychology A1*	*Psychology A1*	*JC AMB C09/10*			
			Time →		*Mon 13-15*	*Tues 11-13*	*Thurs 9-11*	*Fri 13-14*	*Wed 9-11*	*Wed 11-13*	*Fri 14-16*	*Fri 16-18*
25-Sep	1				*No teaching*							
02-Oct	2	***1***			Design Principles C part 1: VSCode and Hello World	Getting started with C	Laying the Foundations	Laying the Foundations				
09-Oct	3	***2***	Lab 1 programming intro (5%)		C part 2: Operators, printf/scanf and conditional statements	C part 1 & 2	Comp architecture; digital signals (parallel); digital i/o;	Comp architecture; digital signals (parallel); digital i/o;			*Collect kit (group-3)*	*Collect kit (group-4)*
16-Oct	4	***3***			C part 3: Loops, arrays and functions	C part 2	Counter-timers; digital signals: serial protocols	Counter-timers; digital signals: serial protocols				
23-Oct	5	***4***		Lab 1 programming submission Thurs 27 Oct (5%)	C part 4: Memory and pointers	C part 3	Sequences, state tables, finite state machines	Sequences, state tables, finite state machines				
30-Oct	6	***5***			C part 5: functions using pointers	C part 4	Analog signals, data acquisition: aliasing, grounding	Analog signals, data acquisition: aliasing, grounding				
06-Nov	7	***6***	Software project prep intro (5%)		C part 6: structures; projects	C part 5	Data conversion including PWM; sensors	Data conversion including PWM; sensors	Lab-1 (group-1)	Lab-1 (group-2)	Lab-1 (group-3)	Lab-1 (group-4)
13-Nov	8	***7***		Lab 1 comprehension quiz Thurs 16th Nov (7.5%)	C part 7: numbers, enums and conditional compilation	C part 7; project	Motion Control: Servo Motors, closing the loop	Motion Control: Servo Motors, closing the loop				

Outline of the Module (2)

20-Nov	9	***8***		Software project prep submission Tues 21st Nov (5%)	Command line arguments and code optimisation	C part 8; project	Stepper motors; drivers; Bresenham and ramping	Stepper motors; drivers; Bresenham and ramping				
27-Nov	10	***9***			Software best practice	Project	Stepper motor dynamics. Solenoids, pneumatics, hydraulics.	Stepper motor dynamics. Solenoids, pneumatics, hydraulics.	Lab-2 (group-1)	Lab-2 (group-2)	Lab-2 (group-3)	Lab-2 (group-4)
04-Dec	11	***10***		Lab 2 comprehension quiz Thurs 7th Dec (7.5%)		Project	Interrupts and real-time issues; FPGAs	Interrupts and real-time issues; FPGAs	Robot Testing (15 min slots)	Robot Testing (15 min slots)	Robot Testing (15 min slots)	Robot Testing (15 min slots)
11-Dec	12	***11***		Software project submission Thurs 14th Dec (20%)	Consolidation and revision	Project	Consolidation and revision	Consolidation and revision	Robot Testing (15 min slots)	Robot Testing (15 min slots)		
18-Dec	13											
25-Dec	14											
01-Jan	15											
08-Jan	16											
15-Jan	17		Exam 55%									
22-Jan	18		Exam 55%									

Outline of the Module (3)

- Lecture notes
  - Notes are in the form of a mini book on C (so no book to buy !) available on Moodle
  - You might also like to review some of the excellent on-line courses
    - http://www.tutorialspoint.com/cprogramming

Lab exercises

- Exercises corresponding to the chapters in the book (use the computer labs to tackle these with help on hand!)

Sample code

- Available on GitHub https://github.com/louisepb/VSMechatronics
  - Folders starting ‘CL’ give code used during the lectures
  - Folders starting ‘C’ give code used in the book and computer lab exercises

Books

- If you do wish to get a book for the course there are a few ones you might like to consider
  - Paul Deitel and Harvey Deitel, C How to Program, 8th Edition, Global Edition, Pearson Education Ltd.: London, 2016, ISBN 13: 978-1-292-11097-4.
  - C Pocket Reference; Peter Prinz, Ulla Kirch-Prinz;ISBN: 9780596004361 Publisher: O'Reilly Media, Inc,
  - C Programming in Easy Steps; Mike McGrath; ISBN: 9781840783636
  - C for Dummies (v.1); Dan Gookin; ISBN: 9781878058782
- However
  - A book is not essential and I would strongly recommend that you browse the many books on C available and choose one that suits you
  - The work we will be covering is ‘basic’ C and will be covered in any good book on the subject

Why study programming?

- A few reasons
  - Pass the module
  - Complete the project
  - Be able to do modules in later years
- The above are true
  - But are not really persuasive arguments
- Remember:
  - We are training you to become highly employable engineers

But why does an engineer need this?

- Many of the systems designed contain embedded microprocessors/control systems

- It is not computer programmers that develop the code
  - It is often the engineers that do this as they best know the systems!
  - We can relate code to the processor and even the hardware on which it runs

SpaceX Falcon 9 (2021)

Examples

- Examples of this are past students who work on & develop the code for:
  - Control systems for Airbus/Boeing
  - Next generation power generators for Rolls-Royce
  - Power convertors for renewable energy systems
  - Map the interaction of EM radiation with people/planes/equipment etc.

Before we move on…

- This is both a **PRACTICAL PROGRAMMING** course
  - It will teach you the basics (hopefully well!)
    - There is syntax to learn (VSCode helps here)
    - Concepts such as loops/decisions
  - You will then have the building blocks to write even more ‘complex’ code
  - It is a skill that develops with practice – and you will get plenty (in the associated labs & the project module)

- And a **SOFTWARE ENGINEERING** course
  - We look at how to solve problems and design robust code
  - The KEY thing is to learn to ‘think’ about the code before writing it – in the same way one would ‘design’ a house before building it
  - We look at what code is – regardless of the language being used

When developing code

- We have a choice of
  - Machine Code
    - The ‘language’ of the processor.
    - When ‘crafted’ can be better optimized than any compiler
    - Can be VERY time consuming to develop – often a highly specialised skill
    - Code is (generally) processor dependent and so not ‘portable’
  - High Level Languages, e.g. APL ,Pascal, C, C++, ADA, Fortran, Algol, COBOL, Python
    - Each suited to different tasks
    - All essentially the same,
      Syntax & keywords of the language that differentiates them
    - Code is ‘portable’
      The compiler sorts things for us

Compilation Vs Interpretation

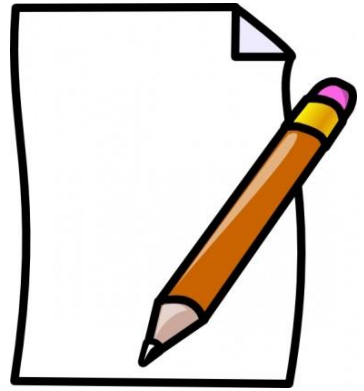
We also have the option (language specific) of

- Interpreted
  - Line decoded & interpreted at run time -SLOW !
  - Program errors often only found at run time
  - Now coming back into use (web based languages such as ASP & PHP)
- Compilation
  - Program Analysed -> Object code
  - Linking Stage -> Executable

The ‘Tools’ of the trade

Creating flowcharts

- To design our code
  - diagrams.net
  - This is a free online tool that allows for the creation of simple (or complex) diagrams
    - https://app.diagrams.net/
  - Or
    - Pen & Paper ☺
    - (Not for project submissions)



Version control

- GIT: To keep our code ‘safe’
  - A free version control system
  - It allows us to keep version of the code so we can ‘go back’
  - We can ‘branch’ code to try things
  - Share code with others who can then ‘check in’ code when they have finished with it
  - https://git-scm.com/downloads

Does this look familiar?

CodeFolder:

- TestCode.m
- TestCode_data_set1.m
- TestCode_data_set1_v2.m
- TestCode_data_set1_v2_with_output.m

The ‘simple’ manual solution

If we were a single developer working on completely separate code projects we could

- Keep the code in its own folder
- Make regular backups (dated & in multiple locations)
- Have plenty of comments to highlight what changed and when

The ‘downside’

- This does however stop the use of common code (which does make things easier)
- It is difficult to see which files changed between backups (at best we can use the date edited)
- We will be making copies of lots of files that have not changed
- We have to ‘remember’ to do backups

The better solution

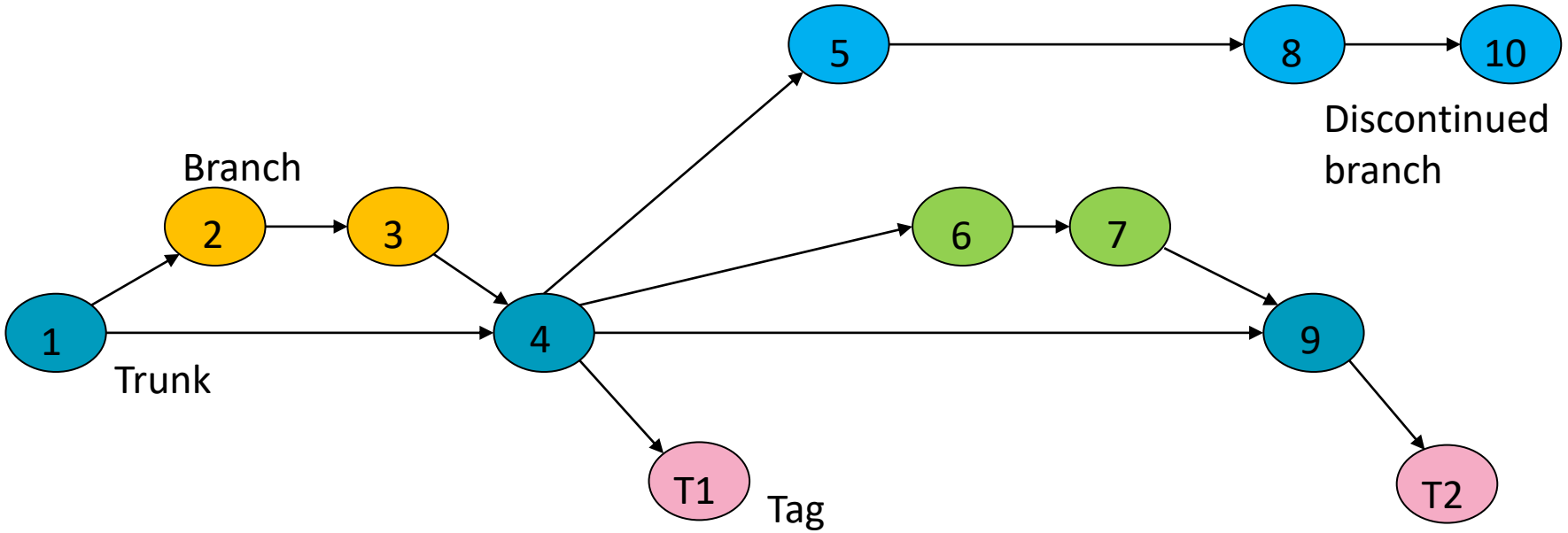
Use version control, eg Git, Subversion or Mercurial

- Keeps track of code changes
- Provides backup if used in conjunction with a hosting service such as GitHub (github.com) or Bitbucket (bitbucket.org)
- Git can be downloaded from https://git-scm.com/

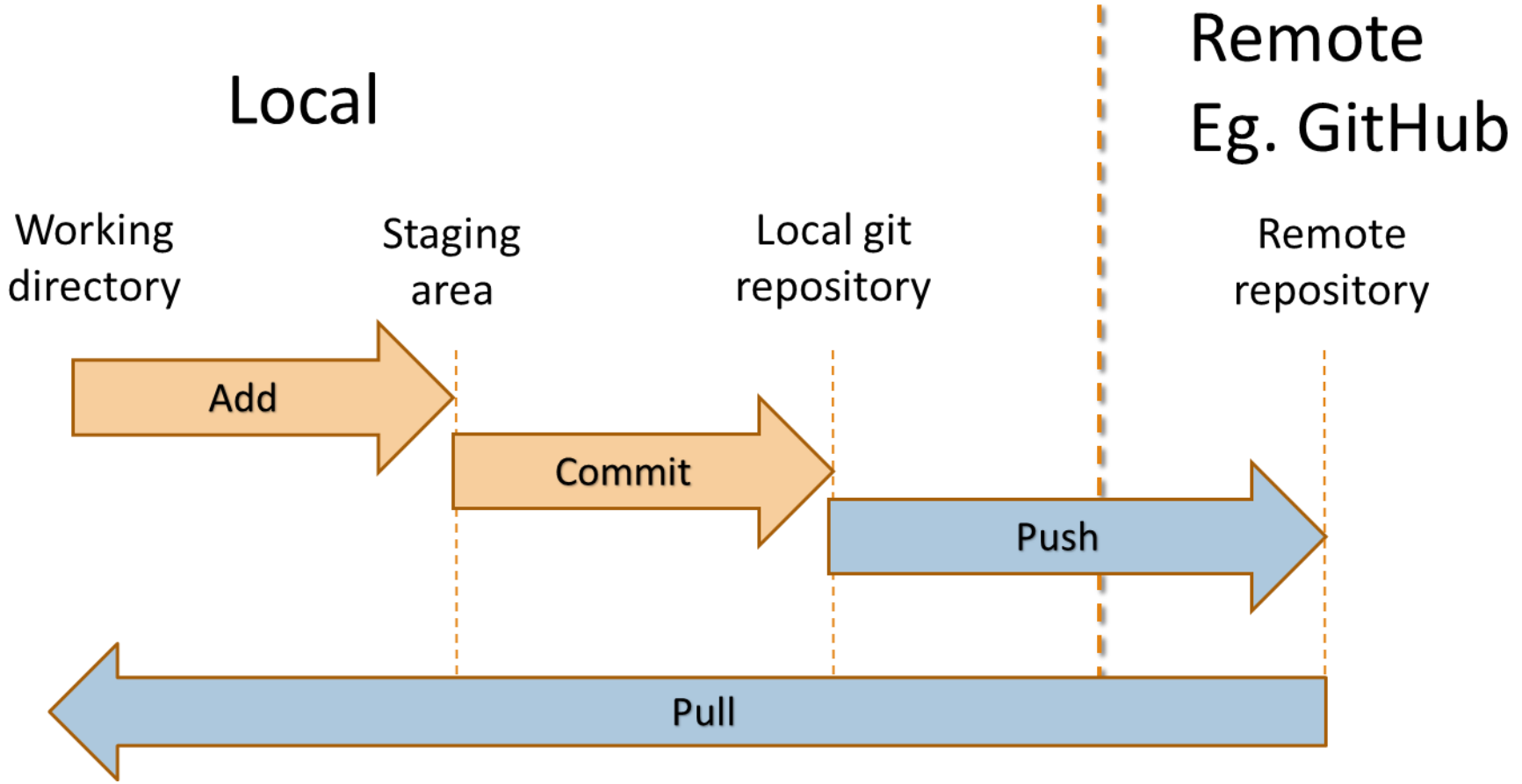
- GIT [1]
  - Git (/gɪt/[6]) is a version control system that is used for software development and other version control tasks. As a distributed revision control system it is aimed at speed, data integrity and support for distributed, non-linear workflows.[10] Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.
  - As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server. Like the Linux kernel, Git is **free** software distributed under the terms of the GNU General Public License version 2. [1] https://en.wikipedia.org/wiki/Git_(software)

Local git workflow

- Distributed system – have own version of the repository on local computer
- Using a remote repository gives backup and easier sharing between developers
- Integrated into some IDEs eg Visual Studio and Matlab
- Easy use of branches for experimental code development



Git workflow



- If you were using a command line, the basic commands to manage an existing git repository are
  - git pull — Gets the latest version of the code
  - git init — Creates a new repository
  - git add . — Adds all changes to local git version (the ‘dot’ means all)
  - git add newfile.c — Only add the file ‘newfile.c’ to the local git version
  - git commit –m “fixed a bug” — Commit the code, adding a comment about the changes made
  - git push — Upload the new version to the server
- Others would then do a ‘git pull’ and get the new version (being prompted for conflicts as appropriate)

Chapter 2

Designing Code

But before we do any coding

- We need to ‘design’ code
  - In the same way an architect would design a building
    - They gather the requirements
    - Consider the limitations, materials available, environment etc.
    - Design plans at the macro and micro scale
    - Others then ‘build’ and test against these plans

- Coding is the same…
  - It is just we use software engineers and software architects!

What is involved in creating software?

- What are the things you need to consider to create a piece of software and/or a software product?
- What are the steps in the process?

- Add your ideas to the padlet:

https://padlet.com/louisebrown7/overview-of-a-software-project-ttiklf86efk760zq

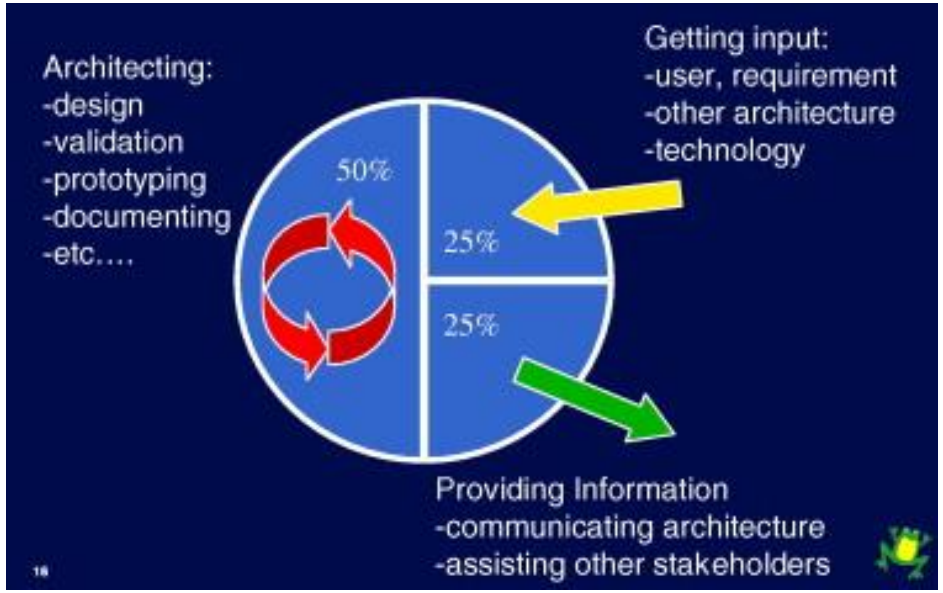
Overview of a software project

- What's involved in creating a piece of software?

  - Requirements gathering
  - High level design
  - Low level design
  - Development
  - Testing
  - Deployment
  - Maintenance

What is a Software Architect?

- Software architects should design, develop, nurture, and maintain the architecture of the software-intensive systems they are involved with.[1]



[1]Kruchten, P. (2008). "What do software architects really do?" Journal of Systems and Software **81**(12): 2413-2416. https://www.sciencedirect.com/science/article/pii/S0164121208002057

Software Architect or Software Engineer?

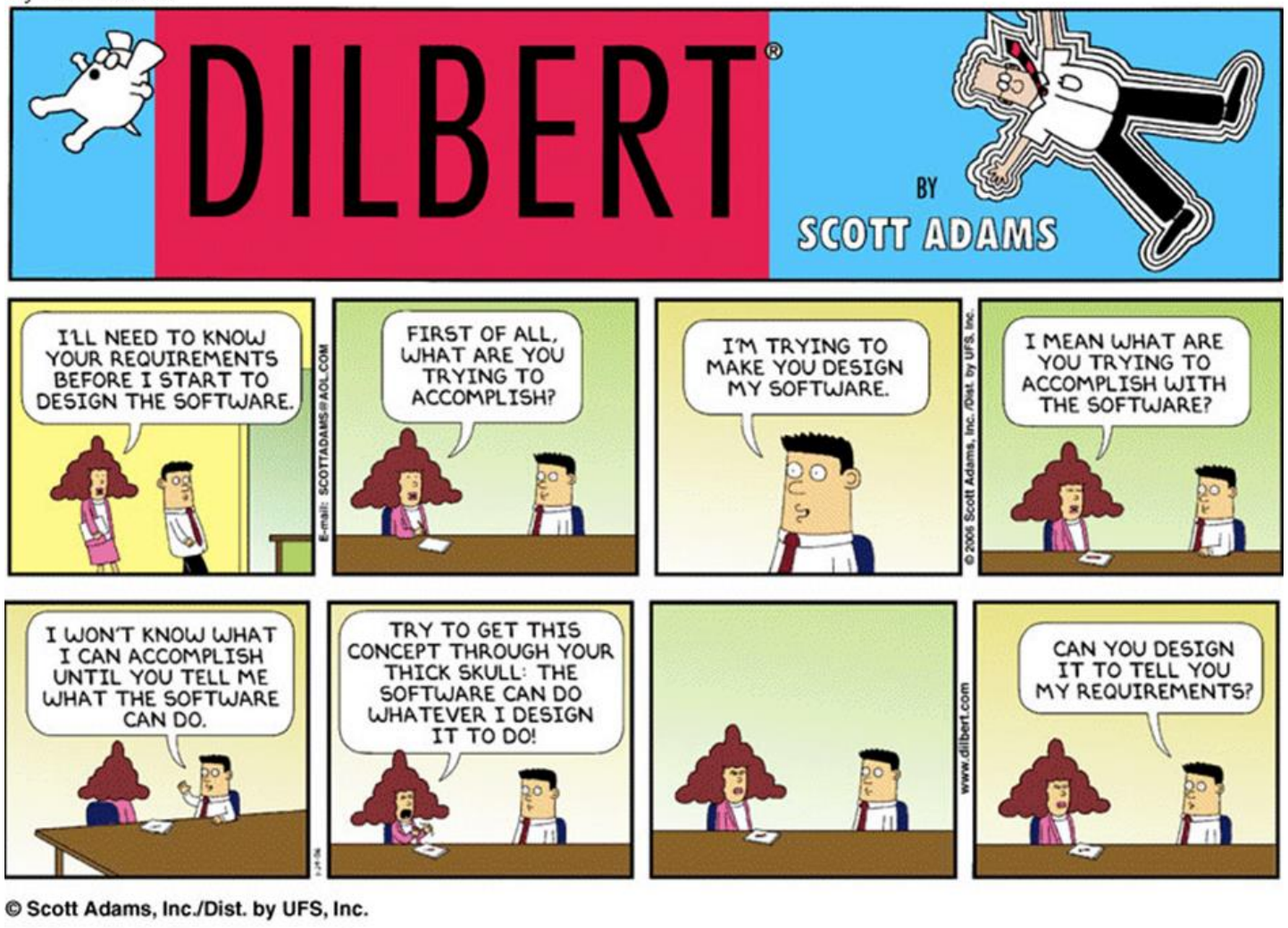
- Software architecture shows the system's structure and hides the implementation details, focusing on how the system components interact with one another.
- Software design concentrates on the system's implementation, often delving into significant detail.
- Software design centres on the selection of algorithms and data structures, as well as the implementation details of every single component[2]

[2] https://medium.com/@concisesoftware/whats-the-difference-between-software-architecture-and-design-b705c2584631

Understand the problem!

Dilbert

by Scott Adams



In short…

Software architects and engineers

- Look at the problem to be solved (often visiting and talking to people) and so gather the requirements
- Consider the limitations, environment etc.
- Designs how the code will function
- Provides test criteria to confirm correct operation (and ‘error’ cases)

This is often the ‘hardest’ and most time consuming part

- But the one that must be done correctly
  - Programmers will work from the plans and develop code accordingly
  - They will not question ‘why’, they just ‘do’
  - You cannot blame the programmer if the design is wrong
    In the same way you cannot blame a builder if the building plans are incorrect

Once the design is done

Programmers

- Take the ‘plans’ and, using their skills, write the code

Testers

- Take the code and test it both at function and system level

You will learn to be all of these

Software Engineer

- Developing the ‘flow’ of the code and developing test scenarios to check it works correctly

Programmer

- Take the ‘plans’ and, using YOUR skills, write the required code (in the appropriate language)
- Document & maintain the code

Tester

- Take the code you have developed and test it both at function and system level

The ‘hard’ part

Learning how to plan the code

Computers (and programmers) take things literally

A woman asks her husband, a programmer, to go shopping.

Wife: “Dear, please, go to the nearby grocery store to buy some bread. Also, if they have eggs, buy 6.”
Husband: “O.K.”

Twenty minutes later the husband comes back bringing 6 loaves of bread. His wife is flabbergasted.

Wife: “Dear, why on earth did you buy 6 loaves of bread?”

Husband: “They had eggs.”

The ‘easy’ part

Coding

- Though at first you may not believe me, is easy ☺
- It is writing a series of statements that will be executed exactly as you have written them

The ‘evil’ part

Testing

- Testing is essential!
  - At ‘functional’ level (think here of components of a system)

    We want to ensure the individual bits work before we start to assemble them
  - At system level (the final working project)

    Does the system work as expected
- We test
  - As we write
  - When we have finished
  - After any changes are made

What you learn is portable

Once you can program in one language it is easy to learn more

- In fact, once you know two/three you can generally fix code in a language you do not know!

This is because once you learn how to think like a programmer

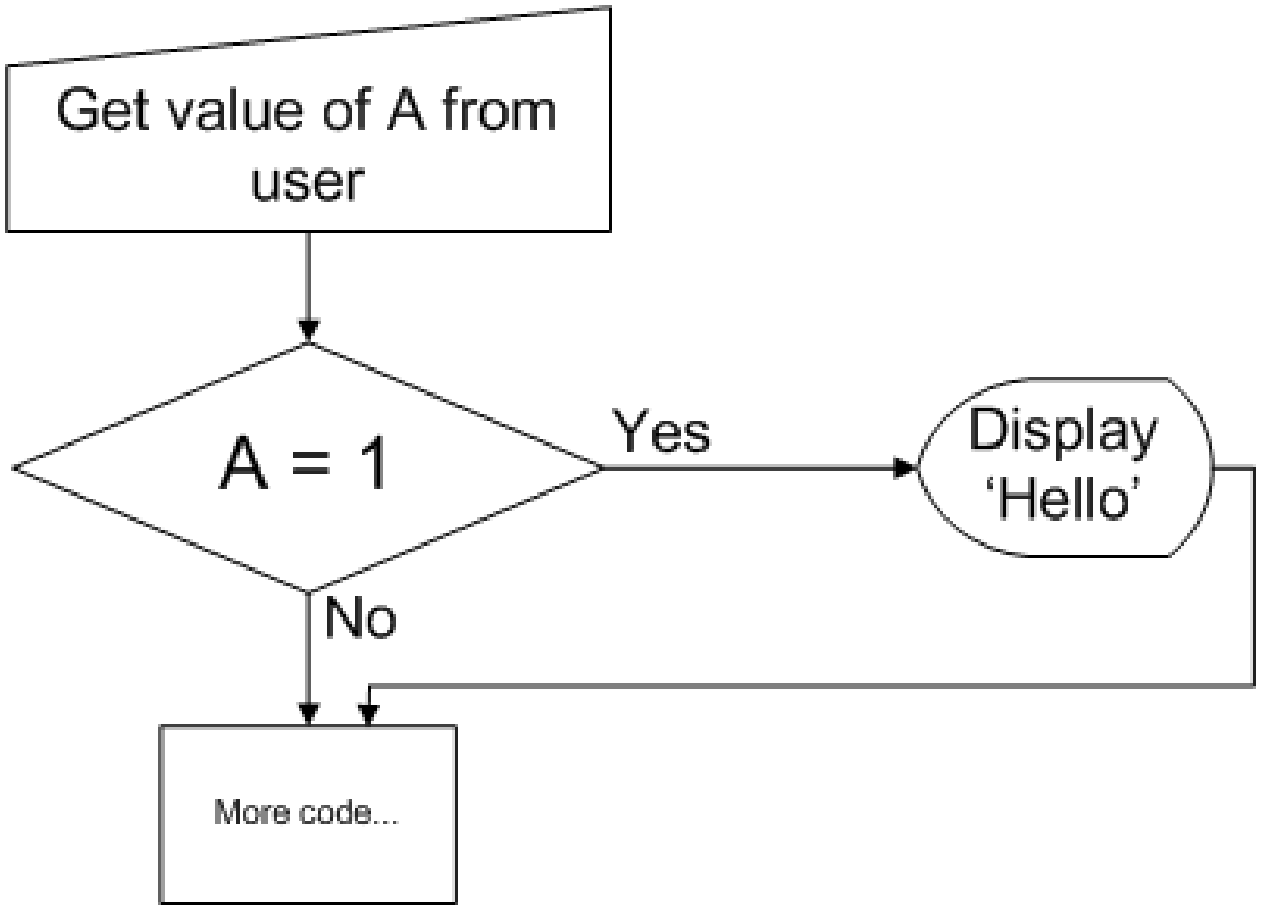
- All you need is the syntax for the new language

And by way of proof…

- In all programming languages there exists
  - If
  - If / else
  - If / else if / else

Pictorially (a flow chart)

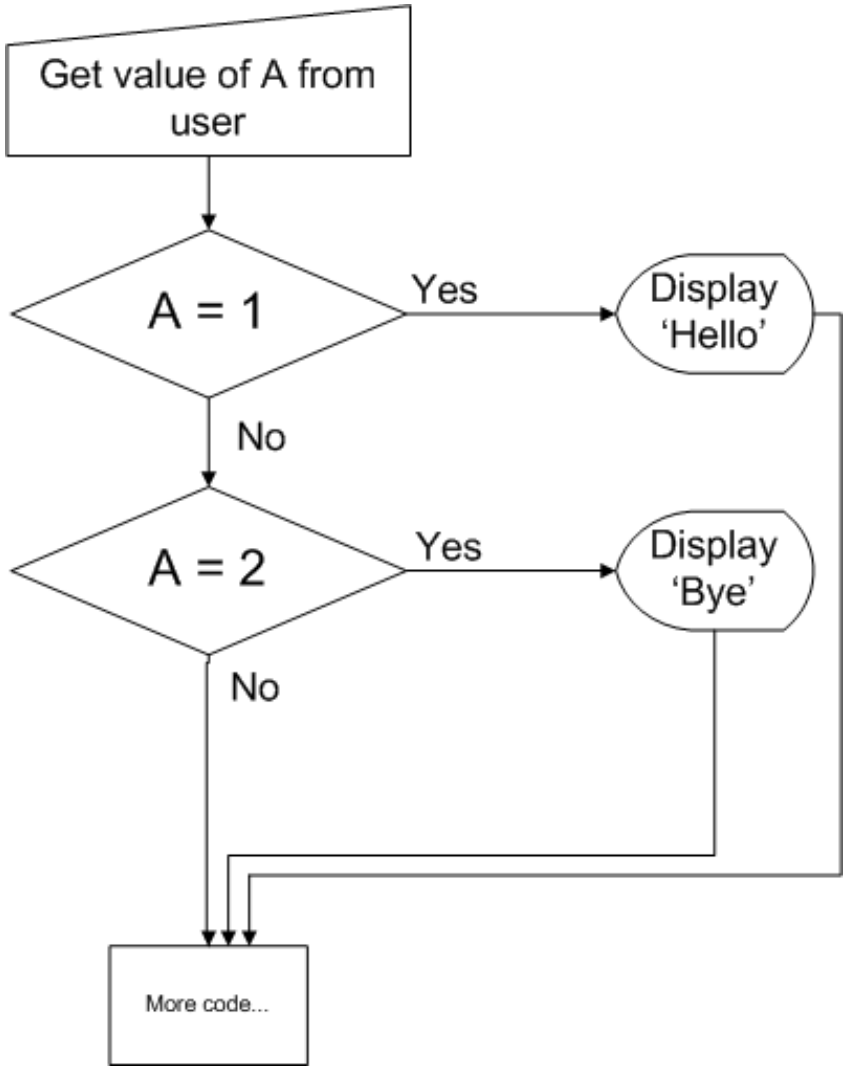
If



Pictorially (a flow chart) (2)

If

Else if

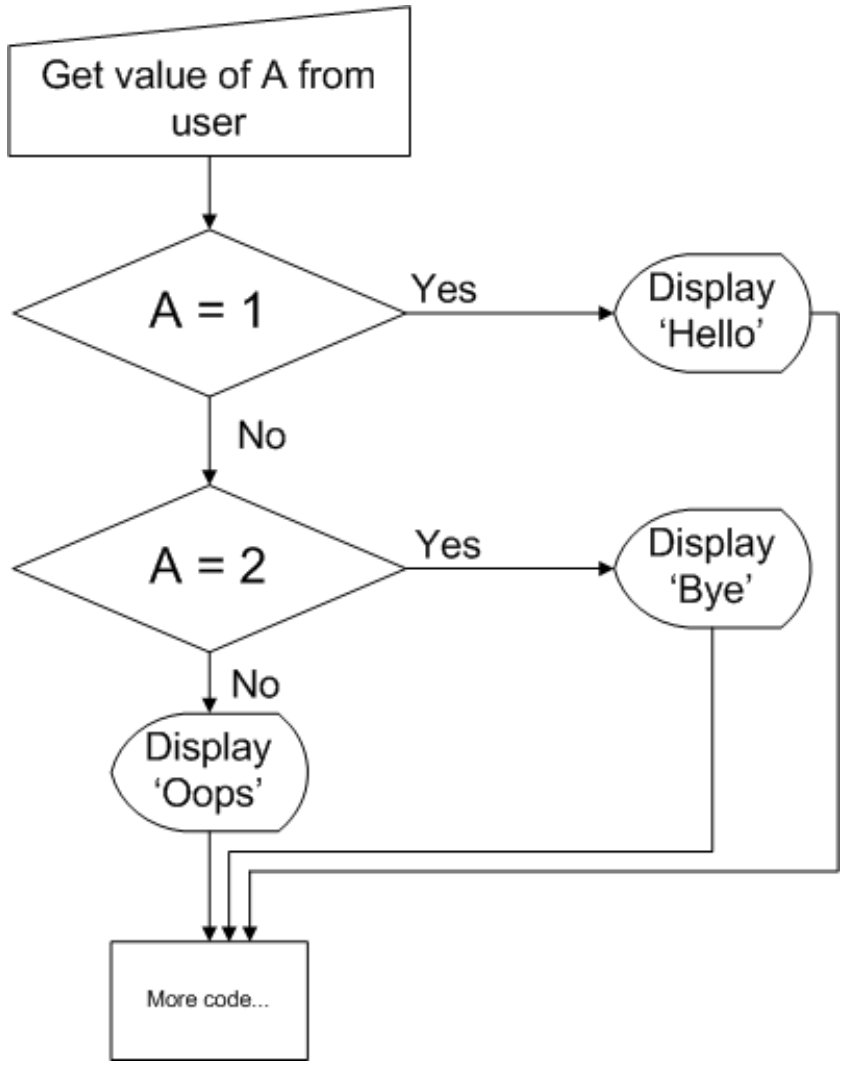


Pictorially (a flow chart) (3)

If

Else if

Else



In code form: if / else & if / else if / else

if c then b1 else b2	CoffeeScript, F#, Haskell, merd, OCaml, SML
if c then b1 else b2 end	Eiffel, Lua, Ruby
if c then b1 elseif c2 then b2 else b3 end	Eiffel, Oz
if (c) then b1 elseif (c2) then b2 else b3 end	Dylan
IF c THEN b1 ELSIF c2 THEN b2 ELSE b3 END	Modula-3
If c Then b1 ElseIf c2 Then b2 Else b3 End If	Modula-2
if (c) b1 else b2	Awk, B, C, C#, C++, Java, JavaScript, Pike, YCP
if c b1 elsif c2 b2 b3	Tcl
if c then b1 elseif c2 then b2 else b3	Tcl
if c then begin b1 end else begin b2 end	Pascal
if c b1 eif c2 b2 else b3	Pliant
if c then b1 elif c2 then b2 else b3 end if	Maple
if c; then b1; elif c2; then b2; else b3; fi	BourneShell
if c; b1; else b2; end	FishShell
if c1, b1, elseif c2, b2, else, b3, end	Matlab

Source:
http://rigaux.org/language-study/syntax-across-languages.html

But the one key thing to remember

- Computers are NOT intelligent
  - They will do exactly what you tell them to

- The ‘trick’ is to:
  - Be specific in what you want the code to do
  - Make NO assumptions

Consider a practical case

- Consider this (based on an old, no longer used, progression rule)
- A student has the following marks

Module1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7	Project
50%	70%	65%	68%	63%	55%	70%	80%

Average = 65.1%

- Rule:
  - If a student has an average of 68% or 69% and half their modules have a mark over 70% or their project mark is over 70% they get a 1st
- Question:
  - Does the student get a 1st class degree?

In a more memorable format

- The following is an example of doing exactly what you are told to

As a good piece of code would!


If your DOG
does a POO
Please put it
in a litter bin.
Please help keep our
open spaces clean.
Signspotting.com

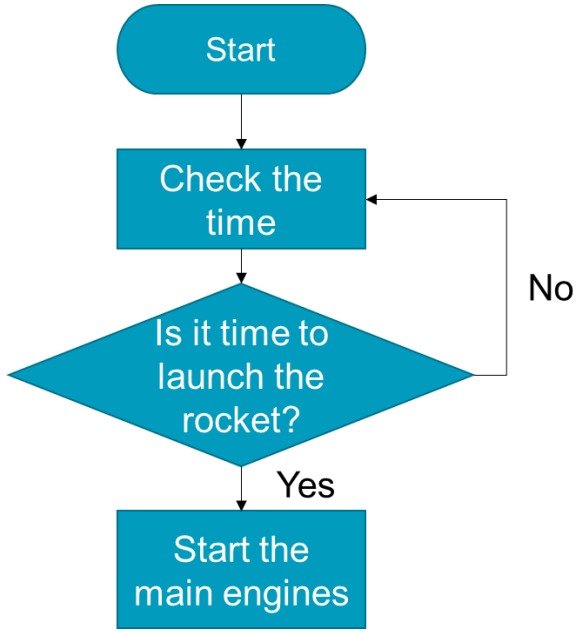
So let's design some code…

- The simplest method is a flowchart (which draw.io is great for!)
  - There are a number of symbols however the most commonly used ones are as below

Symbol	Name	Function
	Start/end	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/Output	A parallelogram represents input or ouptut.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

https://www.smartdraw.com/flowchart/flowchart-symbols.htm

This simple program decides if it is time to launch a rocket...



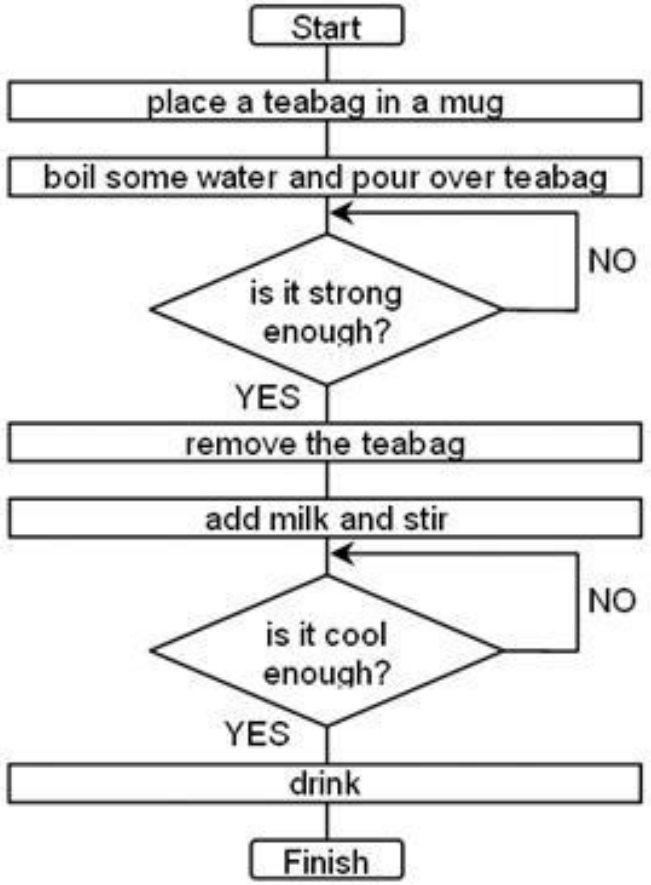
Flowcharts

We can apply the process to many tasks:

Sketch out a flowchart to make a cup of tea

Flowchart for making tea

- This is a very simple example however it is missing a large number of steps!
- What are they?



http://whyhavebusinessanalysis.wordpress.com/2011/05/03/i-just-love-business-process-modelling-bpm/

Exercise

- Sketch out a comprehensive flowchart for a program to solve a quadratic equations where the roots are **not** complex

- Develop some test values (good and bad cases)
  - Assume here users are idiots (This is always a good plan!)
  - Use a table to set out test data
  - Make sure your test data covers all routes through the flowchart

Function	Test Case	Test Data	Expected Output

    - You will be drawing this up in draw.io in the computer lab session

Lab Work for this Week

Taking a problem

- Analysing the problem
- Generating the flowcharts (in draw.io)
- Developing test data
  - Both pass and fail cases

Chapter 3

Hello World

Overview

- Getting Started
  - Visual Studio Code, VSCode – Programming environment
    - What it is
    - Why do we use it?

- Looking at code – the most basic program
  - The structure of a program
  - The basic syntax of C

The programming environment and compiler we will use

We use VSCode

- This is actually a ‘container’ for programming in various languages
- It is available on the Engineering Virtual Desktop
- Quick and easy to get started with
- Free!

It can be downloaded from

- https://code.visualstudio.com/download
- You will need to install extensions for C programming

You may need to install the gcc compiler

- Windows: Use MSYS2 https://www.msys2.org/ to install MinGW-x64
- Linux should already have gcc installed
- MacOS should have CLang installed

- **Full instructions for installation are given in Appendix A of the course book and in the ‘Setting up VSCode for Compiling C Code’ document on Moodle**

Let’s get coding!

- All C programs (in fact code in almost all languages) consist of the same basic parts
  - Pre-processor commands
  - Functions
  - Variables
  - Statements and expressions
  - Comments
- Let’s look at an example – the classic ‘Hello World’ program

The *Famous* "Hello World" Program (1)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

These lines instruct the compiler to 'include' the contents of the files 'stdio.h' and 'stdlib.h'

This is done by the compiler **before** the code is compiled

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (2)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

All C code starts execution at the line

int main()

(regardless of where it is in your code)

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (3)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

Brackets {} are used to block lines of code

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (4)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

Comments are vital to good coding – they allow information to be included within code

Between /* and */
or following //

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (5)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

printf is a function within C that allows us to write text to the display.

In C, the parameters for a function are placed within brackets ( ) , multiple parameters are comma separated.

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (6)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;
}
```

Each statement in C is terminated with a semicolon ;

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (7)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

The last statement

return 0;

Terminates the main() function – so ending the program

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

The *Famous* "Hello World" Program (8)

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

int main(void)
{
    /* My first program in C */
    printf("Hello World \n");
    return 0;                          // Return from prog
}
```

Time to see the code in action!

Note: Writing 'Hello World' is a tradition in programming – there are whole web sites dedicated to it, e.g.

http://helloworldcollection.de/

C3\hello_world.c

Now it’s your turn

Try this before the computer lab on Tuesday

- Make sure you can access VSCode
- Create a Hello World project
- Build and run the program
- Locate the project files on your computer and see what files have been created by the build

Chapter 4

The Very Basics of C

The syntax of C

- As with learning any language (programming or spoken), there is the grammar and syntax to get to grips with
- Initially you may find this (somewhat) infuriating as a simple typing mistake can seem to hold you up
- One thing you will need to watch out for is when to (and importantly, when NOT TO) use semicolons

The syntax of C (2)

- Lines of code are terminated with a semicolon
- A semicolon on its own is a valid null statement
  - This can cause problems in certain circumstances)
- Indentation
  - helps
    - Readability

      And

      Must

      - Be
        - Used!
- Blocks of code go in curly brackets/braces {..}

Brackets {}, There are two styles…

There are two types of people.

```
if (Condition)
{
    Statements
    /*
     ...
     */
}
```

```
if (Condition) {
    Statements
    /*
     ...
     */
}
```

Programmers will know.

Comments!

- Single lines of comments can be prefixed
  - ***//***
  - E.g. // This is a comment

- Block comments go between
  - /* and */
  - /* Comments are ESSENTIAL */

- A lack of comments will cost you marks!

Identifiers: Naming things…

- An identifier is the term we use for something in code that WE define (a ‘user-defined’ item).
- The rules on naming are
  - They **can only start** with a letter **A** to **Z**, **a** to **z** or an **underscore** ‘_’, optionally followed additional letters, underscores or digits
  - You **CANNOT** use punctuation characters (@, $, % etc.)
  - Some valid examples would be
    - abc
    - _temp
    - i
    - myname
- Note that C is a **case sensitive language** so
  - **Age** and **age** are two different identifiers in C

Write for Humans

- Make names meaningful and distinctive
- Avoid hx, hy – use HeightX and HeightY
- Avoid names that are very similar eg results and results2

White spaces costs nothing and makes code much easier to read

Code should be readable!

Types of Variables

- When programming we often need to store values.
- To do this we define variables (using an identifier that ideally indicates what is being stored)
- There are many types of variables in C, for now we will consider the most basic type – those that hold numerical values
- We then consider the type of number they will hold
  - Integer (whole numbers)
  - Floating point (those numbers that may have a decimal part)
- We also need to consider the size of number that will stored…

Integer types

There are many integer types. Below are a few (arrows highlight the ones we will most often use)

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Note: These are typical values, the storage size and range may differ on some systems, we will learn how to get these for systems in later lectures

Floating point numbers

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Note: These are typical values, the storage size and range may differ on some systems, we will learn how to get these for systems in later lectures

Use of Variables

To create a variable we specify the type then the variable(s) to be created

- To create multiple variable of the same type separate them with commas
- **Note**: If no initial is given the value in the variable is **UNDEFINED** ← Take note!
- Do not make the mistake of assuming it will be zero!
- It is not a bad idea to give every variable a default value

```
int   a, b, c, sum;      /* Define integers – no initial value*/
int   d = 0;             /* Define & set initial value */
```

Use of Variables (2)

Also

- Ensure you remain within range
  - C won't check this for you, but may give warnings
- Use informative variable names
- Be very careful when 'number crunching'
  - Common mistake

    a=3, b=5 : if a and b are both integers a/b = 0!

    This can then give a 'divide by zero' error later

Use of Variables (3)

- Likewise be careful of mixing variable types
  - When mixing integers/floats be careful as the result may not always be of the type you expect
- Basically
  - Consider the problem and then from this determine the type of variable to use!
- And, if required/appropriate, make use of typecasting

Typecasting: A solution to the previous points!

Consider

- Variables A & B are integers with a=10, B=3
- Variable C is a float

- For the calculation
  - C = A / B
- We would expect the value in C to be 3.3333 however as A & B are integers the calculation is done as integer mathematics (giving 3) which is then stored in C
- The fix is to ‘typecast’ a variable to another type by putting the ‘temporary’ type in brackets before the variable – it is then treated as this type for the purposes of the calculation, assignment etc.
- So, to get the correct answer to the above calculation we typecast A & B as floats, as below

  - C = (float)A / (float)B;

C4\wrong_answer.c C4\correct_answer.c

Mathematical Precedence

Order in which calculations are performed

- 1st Function calls, Brackets & operators
- 2nd Multiply, divide and remainder
- 3rd Addition and Subtraction

A simple mnemonic is

BODMAS

- Brackets, operators, divide, multiply, add, subtract

Mathematical Precedence (2)

E.g.

X * Y * Z + A / B - C / D

Can be written (and is calculated as)

(X * Y * Z) + (A / B) - (C / D)

The brackets are not strictly necessary, but their addition makes the code easier to read

Chapter 5

Output

Displaying Variables (and text)

- It is all very well being able to define variables and use them within code (e.g. for example calculations)
- It is also ‘handy’ to be able to display their values on the screen
- The programming term for getting/displaying information is **Input & Output**
  - We will look at output now – as then we can write code that tells us things (e.g. the result of a calculation)
  - Input is covered in the lecture for chapter 7 of the course book

Displaying Variables (and text) (2)

- The ‘general’ function in C we use do display output is

  printf

- It is a function that can take one or more parameters
  - This is somewhat ‘unusual’ in programming in C where functions generally expect a fixed number of parameters.
- There must be at least one parameter – the text to display

```
printf("Hello world!");
```

Function: printf, used to output to the display

Parameter: The text to be displayed contained in double quotation marks

Formatting Characters

- There are some formatting options for things that we cannot ‘type’ into code (e.g. a ‘new line’)
- The two most common are
  - \n Insert a new line
  - \t Insert a TAB character
- There are more – take a look on-line!
- https://www.ibm.com/docs/en/rdfi/9.6.0?topic=set-escape-sequences

For displaying variables

- To display the contents of a variables using *printf* we use a ‘substitution’ approach
- The 1st parameter in the *printf* statement is still the text to display but within this we use ‘place holders’
- When the code is executed, these ‘place holders’ are substituted with the values stored in variables

Displaying the contents of variables

Variable place holders – replaced (at run-time) with the contents of a variable

- %d Used to display an int (you can also use %i)
- %f Used to display a floating number
- %c Used to display a single character
- %s Used to display a string (of **char**acters)
- %x Used to display in hexadecimal
- %#x Used to display in hexadecimal with 0x in front of number

IMPORTANT!

For things to work correctly

- We **MUST** provide a variable for each place holder
- The variable and place-holder type **MUST** match!

i.e.

- To display an integer we **MUST** use %d (or %i)

This is best shown via an example

An example of formatting and place holders

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int a,b,c,sum;       /* Define variables */
    a = 1;               /* Assign values */
    b = 2;
    c = 3;
    sum = a + b + c ;    /* Calculate sum & Display */

    printf ("\nThe sum of %d + %d + %d is %d \n", a, b, c, sum);

    return 0;            /* Return from prog */
}
```

LC5\printf_example.c

Tidying up output

We can ‘enhance’ the variable format string (%d, %f) to improve how we display numbers

Things that can be specified are

- The number of characters to used to display a value
- Where whitespace will be added
  - Before / after the text to be outputted

Note:

- For numbers if more characters are required than that ‘stated’ in the formatting string, the value is over-ridden
- For strings the output is truncated

Tidying up output (2)

- For integers we can specify the number of characters to use (space will be used to pad)
  - %6d Print as an integer with a width of at least 6 wide, whitespace added at the ‘front’
  - %-6d Print as an integer with a width of at least 6 wide, whitespace added at the ‘end’
- Reminder:
  - If more characters are actually needed (e.g. we specify 4 but the number to display is 123456 the format will be automatically overridden)

Tidying up output (3)

For floats we can specify the number of characters to use in total for the number as a whole (can be omitted) and the precision

- %4f Print as a floating point with a width of, at least, characters 4 wide (precision not specified)
- %.4f Print as a floating point with a precision of four characters after the decimal point
- %3.2f Print as a floating point at least 3 wide and a precision of 2DP

C5\formatting_numbers.c

There are a few others

- Some further examples

%e	64-bit floating-point number (double), printed in scientific notation using a lowercase e to introduce the exponent.
%E	64-bit floating-point number (double), printed in scientific notation using an uppercase E to introduce the exponent.

%x	Unsigned 32-bit integer (unsigned int), printed in hexadecimal using the digits 0–9 and lowercase a–f.
%X	Unsigned 32-bit integer (unsigned int), printed in hexadecimal using the digits 0–9 and uppercase A–F.

A quick on-line search for formatting options in will give you a very long list of options!

Now we know a bit…

As we can now

- Define variables,
- Assign them values and
- Display them on the screen

We can start with some real coding!