ENGG1003

INTRODUCTION TO PROCEDURAL PROGRAMMING

STAFF

- Course Co-Ordinator: Prof. Rick Middleton
  - Email: Richard.middleton@newcastle.edu.au
  - Room: ICT311
  - Consultation: By appointment through Jasmine McGee: Jasmine.Mcgee@newcastle.edu.au
  - Primary contact for final exam adverse circumstances applications

STAFF

- Me: Mr. Brenton Schulz
  - Consultation: EF106, Thursday 10am-5pm
  - Email: Brenton.Schulz@newcastle.edu.au
  - Assume any non-exam problem comes to me
- Lab demonstrators
  - Too many to mention, you'll meet them in labs
  - Mix of postgrad and undergrad students
    - Ask them about their work, future studies, etc!

ABOUT ME

- Massive nerd
  - Who else would teach *programming*?
- Enjoy all types of music
  - Thrash metal, death metal, power metal, pirate metal, black metal
  - Yeah, *all types*
- Enjoy games where you just *shoot everything*
  - DOOM
  - XCOM
  - Serious Sam
  - Quake III
    - Ok, maybe you weren't born yet

BLACKBOARD

- Accessed via: http://uonline.newcastle.edu.au
  - Does anyone use QR codes? Didn't think so, have one anyway.
- All courses upload notes, lecture recordings, announcements, grades, etc. to Blackboard
- Your responsibility to check regularly. Typically daily.

COURSE CLASSES

- Two lectures / week
  - This one (9-11am, RW149 aka "Nursing lecture theatre")
    - Overflow when required, EAG01, up the stairs in Engineering building "A"
  - Friday 9-10am, HD01 aka "Griffith Duncan Theatre "
- A 3hr computer lab. **Starts this week!**
  - You sit at a PC among 20-40 other students and get given tasks to do
  - Tasks distributed on Blackboard, typically a PDF, maybe template code
  - One or two demonstrators are paid to be there and answer your questions

TEXTBOOKS

- Same as previous years
- Jeri R Hanly & Elliot B Koffman, Problem Solving and Program Design in C (7th Ed.), Pearson 2013 (ISBN 978-0-13-293649-1)
- Holly Moore, MATLAB® for Engineers (Edn 3), Prentice-Hall 2012, (ISBN 978-0-13-210325-1)
- Not *strictly* required
  - There are thousands of websites that cover C
  - The course notes are written with the aim of being self-contained
  - Buy a 2nd hand copy or access library resources (this includes eBooks via http://library.newcastle.edu.au)

THINKING OF DROPPING OUT?

- Come talk to me! What can be done to help?
- HELP census, 22nd of March
  - Allows you to withdraw without financial or academic penalty
- Withdrawing before the final exam period does not incur academic penalty
  - You still pay for the course though

WHERE TO FIND HELP

- Google
  - Copy/paste error messages
  - Search for C tutorials (there are *lots*)
- Your lab demonstrator
  - Only during enrolled lab times
  - They will help you Google
- Textbooks
- Me, EF106, Thursday 10am-4pm
  - Or via email: Brenton.Schulz@newcastle.edu.au

ASSESSMENTS

- Passing grade is 50%
  - Must also score 40% or higher in the final exam

Assessed Laboratory 1,	Week 4,	5%
Mid-semester quiz,	Week 6,	10%
Programming assignment 1,	Week 7,	12.5%
Assessed Laboratory 2,	Week 9,	5%
Assessed Laboratory 3,	Week 11,	5%
Programming assignment 2,	Week 13,	12.5%
Final Exam,	TBA	50%

ASSESSMENTS

- Programming is quite unforgiving
  - If you develop code on a private machine it may not work on the university computers
  - Assessment demonstration on privately owned laptops is totally fine
  - Uni IT doesn't support Linux but I **wholeheartedly** do
    - Feel free to demonstrate assignments using vim/make/gcc
- All assessments (except the final exam) are held during your lab session
  - Assessment in a different lab session requires approval from me
  - The demonstrators will have class lists

WHAT IS "PROCEDURAL PROGRAMMING"?

- Telling a computer what to do via a list of steps
- Written in a language the computer can understand
  - Ideally, the human writing understands it too
- This course covers the languages "C" and MATLAB.
  - Created in the 1970s isn't C *as old as my parents*? How is this still relevant?
  - Well, lets check a 2018 IEEE survey
- Why not Python?
  - Politics. Python's dominance is quite new and no 2nd/3rd/4th year engineering courses use it (yet).

WHY DO I NEED PROGRAMMING?

- ELEC/MECHA/Computer systems engineering
  - Embedded systems - C
    - Programming small computers in home appliances, UAVs, wireless sensors, IoT, etc
    - You will all do this in ENGG1500 on the "Arduino" microcontroller platform
  - Control systems - MATLAB
    - Designing mathematical models which make a thing do a thing
    - Eg: Car cruise control, temperature control, controlling robot arms, etc
  - Numerical methods – C and MATLAB
    - Catch-all term for any kind of heavy lifting arithmetic done on a PC or supercomputer
    - Applications typically quite specific

WHY DO I NEED PROGRAMMING?

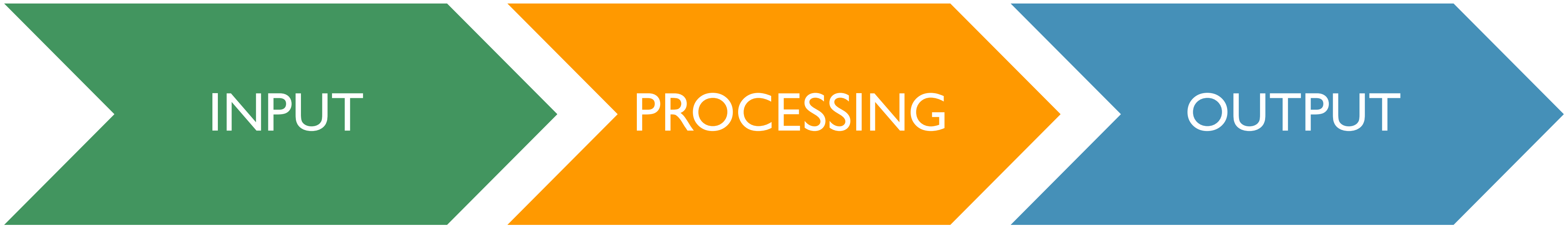
- MECH/CHEM/Medical/Aerospace
  - Many of you will program embedded systems in C
    - MECH students use Arduinos in 2nd year
    - Almost all medical equipment is an embedded system.
  - MATLAB is used all over the place for things I don't understand.
    - Ask your demonstrators in other courses

WHAT IS A COMPUTER?

- How is this relevant to this course?
  - In order to write instructions (programming), you must have a relevant understanding of how computers work
- A Computer is an electronic device designed to perform calculations very quickly
- This seems rather restrictive, just performing mathematics
- But when you consider its other capabilities
  - Speed
  - Communication with other electronic devices (peripherals)
- Then mathematics gives you
  - A word processor, sinus rhythms of a persons heart, how the ailerons should move to bank a plane, how a robot should weld a car body, how much heat is needed to maintain a chemical reaction, weather predictions, etc.

FUNDAMENTAL COMPONENTS

- Fundamental components of every computer



INPUT

- Computers only understand electrical signals
- More specifically those signals represent two states ON or OFF (binary 0 or 1)
- What about a keyboard?
  - It is a device which converts each keystroke into a series of ON/OFF voltages
- What about a mouse?
  - It is a device which converts movements into a series of ON/OFF voltages
- For our model we consider INPUT to be
  - Any series of ON/OFF voltages which the computer needs to perform its calculations
- A device that generates INPUT we will call an INPUT DEVICE

PROCESSING

- PROCESSING is the main function of a computer
- Once the INPUT for a calculation is available, the computer will perform some PROCESSING
- PROCESSING is a series of manipulations performed on the INPUT
  - This involves following a very specific set of instructions
  - The writing of those instructions is called programming

OUTPUT

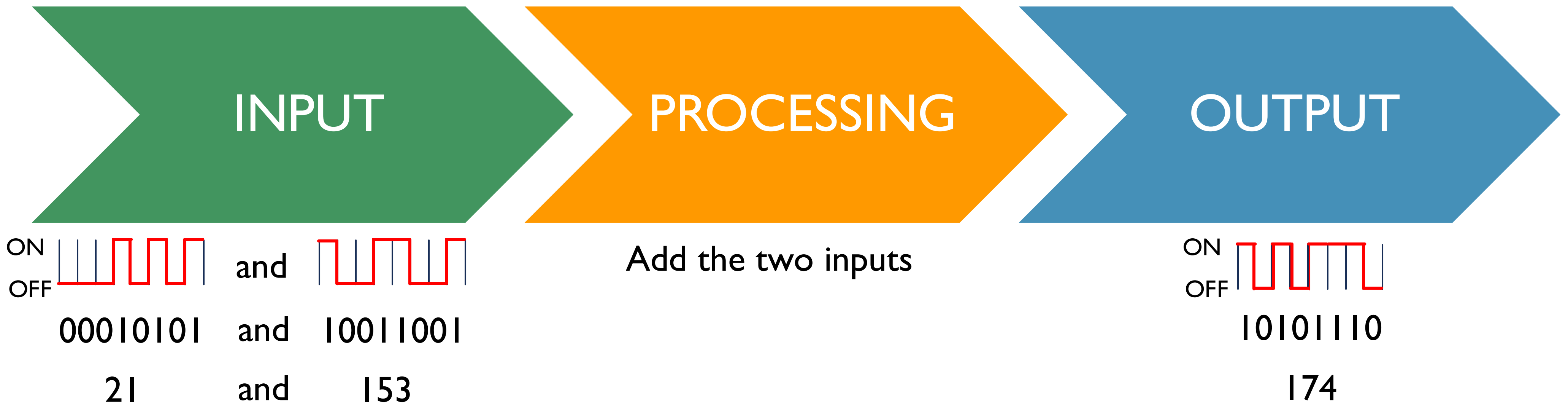
- Once the PROCESSING is complete
- The computer must have a way of presenting the results
- This is the OUTPUT
  - OUTPUT is any series of ON/OFF voltages that represents the results of PROCESSING
- To make the OUTPUT more useful we need an output device

OUTPUT

- OUTPUT DEVICE is any peripheral that takes a series of ON/OFF voltages and manipulates them into something useful
- Examples:
  - Printer
  - Monitor
  - Auto cruise control
  - LCD showing the oxygenation level of a patient’s blood
  - Rudder adjustment in a fly by wire system

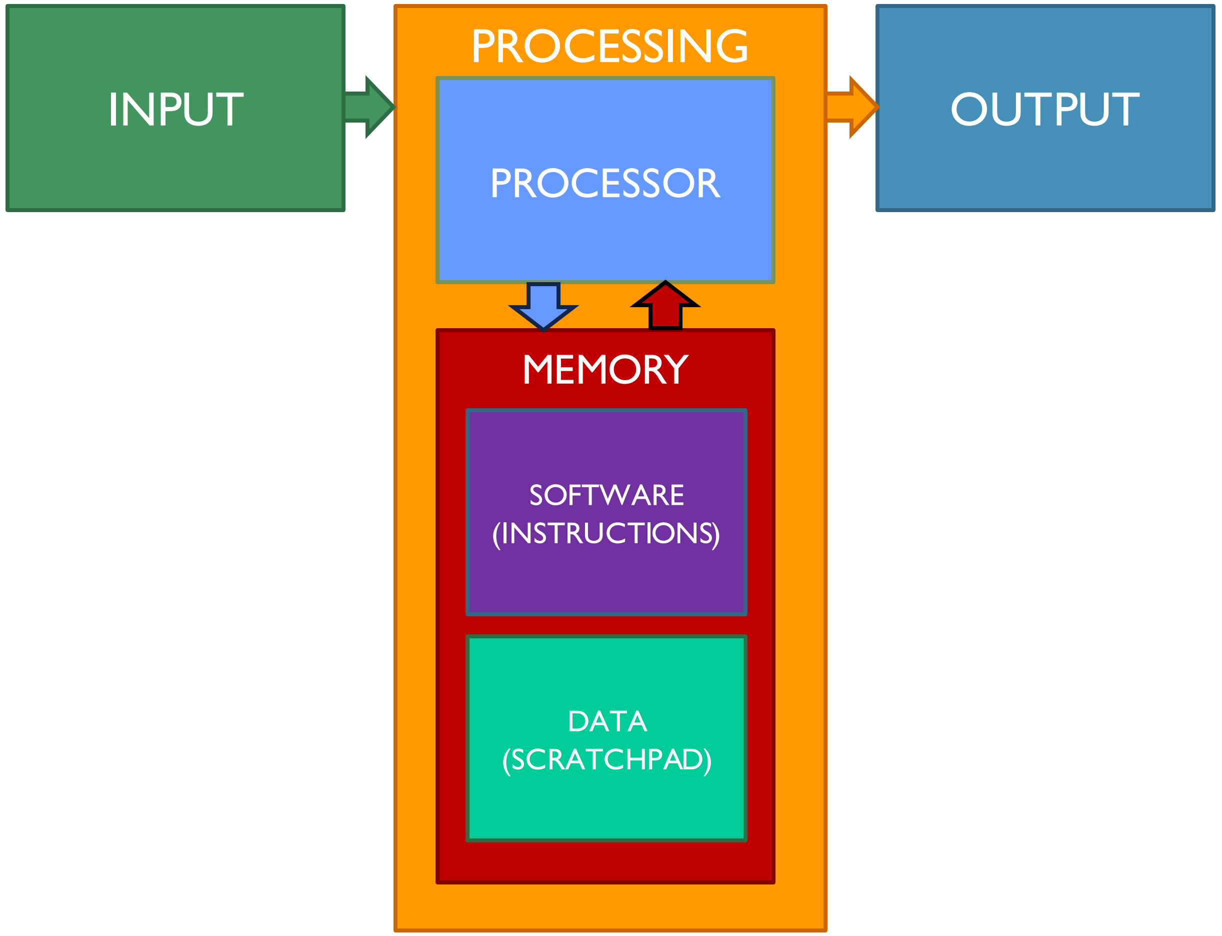
EXAMPLE FUNDAMENTAL COMPONENTS

- Fundamental components of every computer



PROCESSING IN DETAIL

- PROCESSING is complex
- Requires multiple key sub-components to help
- For our purposes we define
  - PROCESSOR
  - MEMORY
  - SOFTWARE



PROCESSING IN DETAIL - PROCESSOR

- PROCESSOR
  - Performs mathematical and data manipulation tasks
  - Has sub-components, but they are not relevant for this course

PROCESSING IN DETAIL - EVERYTHING IS A NUMBER

- Computers can only process numbers
- Things that aren’t numbers need to be represented by them
  - Text: ASCII and Unicode
  - Pictures: each pixel allocated a red-green-blue intensity value
  - Sound: the “waveform” is sampled at regular intervals and stored as a series of numbers
  - (NB: multiple systems exist for all of the above)

PROCESSING IN DETAIL - MEMORY

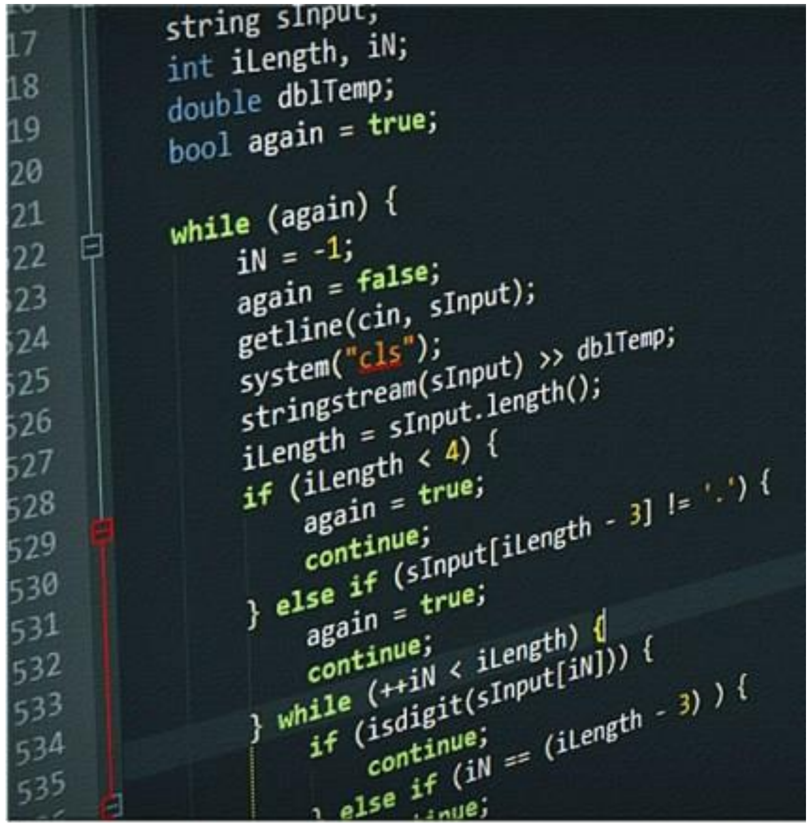
- MEMORY
  - Like humans, computers need to store intermediate results
  - MEMORY acts like a set of written notes for the computer
  - Further relevant subdivision
    - SOFTWARE – Instructions for the calculation
    - DATA – The information required for the calculation (scratchpad)

PROCESSING IN DETAIL - SOFTWARE

- SOFTWARE
  - This is the main event for this course (C or MATLAB®)
  - These are the detailed instructions that the PROCESSOR will follow to perform the desired calculations
  - Instructions **directly** understood by the PROCESSOR are
    - Very specific to each PROCESSOR manufacturer (Instruction Set)
    - Limited in number
    - Simple, e.g.,Add number in memory location 1 to the number in memory location 2 and put the result in memory location 3
    - Again each instruction is encoded as a series of ON/OFF voltages

SOFTWARE

- SOFTWARE for our purpose will be divided into three groups
  - MACHINE code
  - ASSEMBLY language
  - HIGHER-LEVEL languages



```
string sInput;
int iLength, iN;
double dblTemp;
bool again = true;

while (again) {
    iN = -1;
    again = false;
    getline(cin, sInput);
    system("cls");
    stringstream(sInput) >> dblTemp;
    iLength = sInput.length();
    if (iLength < 4) {
        again = true;
        continue;
    } else if (sInput[iLength - 3] != '.') {
        again = true;
        continue;
    } while (++iN < iLength) {
        if (isdigit(sInput[iN])) {
            continue;
        } else if (iN == (iLength - 3) ) {
            inue;
```

SOFTWARE – MACHINE CODE

- The PROCESSOR can only understand MACHINE code
  - Example, one instruction for a x86 based CPU
    - 0110 0110 1000 0011 1100 0000 0000 1010
    - Difficult for humans to understand
  - Very PROCESSOR specific
    - Will not be understood by another PROCESSOR

SOFTWARE – ASSEMBLY LANGUAGE

- ASSEMBLY Language
  - Uses simple mnemonics to describe the purpose of the instruction
  - Example, one instruction for a x86 based CPU
    - Machine code: 0110 0110 1000 0011 1100 0000 0000 1010
    - Assembly language: `ADD AX, 10`
  - A bit easier for humans to understand
  - Still PROCESSOR specific

SOFTWARE – HIGHER-LEVEL LANGUAGES

- HIGHER-LEVEL Languages (C, MATLAB, Java, Python, C++, FORTRAN,…)
  - Uses more human readable instructions
  - Increases the complexity of each instructions so that common calculations can be done with fewer instructions
  - Example, one instruction for a x86 based CPU
    - Machine code: 0110 0110 1000 0011 1100 0000 0000 1010
    - Assembly language: `ADD AX, 10`
    - In C: `AX = AX + 10;`
  - Much easier for humans to understand
  - Not PROCESSOR specific
  - Allows writing of much more complicated instructions

RECOGNISING COMPUTERS

Device		Specification
Desktop PC		INPUT DEVICE: Keyboard and Mouse OUTPUT DEVICE: Monitor, Speakers PROCESSING: PROCESSOR: Intel i7 64-bit CPU MEMORY: 8GB RAM
Laptop		INPUT DEVICE: Keyboard and Touch Display OUTPUT DEVICE: Monitor, Speakers PROCESSING: PROCESSOR: Intel i7 64-bit CPU MEMORY: 8GB RAM
Smart Phone		INPUT DEVICE: Touch Sensor, Microphone, Accelerometers, GPS receiver, 4G Receiver OUTPUT DEVICE: Display, Speaker PROCESSING: PROCESSOR: Qualcomm Snapdragon 808 ARMv8-A 64-bit CPU MEMORY: 8GB RAM

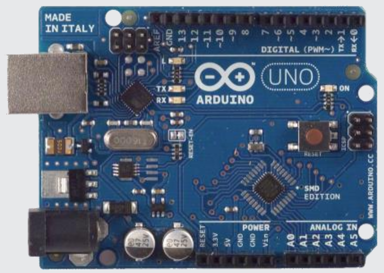
RECOGNISING COMPUTERS

Device	Specification
Raspberry Pi 3+	INPUT DEVICE: Keyboard, Electrical signals (I/O) OUTPUT DEVICE: HDMI Port to Display, USB, Audio PROCESSING: PROCESSOR: Broadcom BCM2837B0 64-bit quad-core ARM Cortex-A53 CPU MEMORY: 1GB RAM
Sony PlayStation 4	INPUT DEVICE: Gaming Controller OUTPUT DEVICE: HDMI Port to Display PROCESSING: PROCESSOR: AMD x86-64 "Jaguar" CPU MEMORY: 8GB RAM
Apple TV	INPUT DEVICE: Remote control OUTPUT DEVICE: HDMI Port to Display PROCESSING: PROCESSOR: Apple A10X Fusion ARM 64-bit CPU MEMORY: 2GB RAM

RECOGNISING COMPUTERS

Device	Specification
Smart TV	INPUT DEVICE: Remote OUTPUT DEVICE: Display PROCESSING: PROESSOR: Dual-core ARM Cortex-A9 1Ghz MEMORY: 1GB RAM
PLC (Programmable Logic Controller)	INPUT DEVICE: Electrical signals OUTPUT: Electrical signals PROCESSING: PROCESSOR: Intel 8051 CPU MEMORY: SOFTWARE: 2KB RAM DATA: 128B RAM
Defibrillator	INPUT DEVICE: Electrical signals from electrodes OUTPUT: Defibrillation current PROCESSING: PROCESSOR: STM32 STM32F429 ARM-Cortex M4 32-bit CPU MEMORY: 256KB RAM

RECOGNISING COMPUTERS

Device	**Specification**
Network Router	INPUT DEVICE: Ethernet, Radio Signals OUTPUT DEVICE: Ethernet, Antennae PROCESSING: PROCESSOR: Broadcom BCM21664T Dual-core ARM Cortex-A9 32-bit CPU MEMORY: 1GB RAM
Arduino UNO 	INPUT DEVICE: Electrical signals OUTPUT: Electrical signals PROCESSING PROCESSOR: Microchip ATmega328 8-bit Microcontroller (CPU) MEMORY: SOFTWARE: 32KB RAM DATA: 2KB RAM

RECOGNISING COMPUTERS

- All these devices are
  - Computers
  - Have the fundamental elements INPUT, PROCESSING and OUTPUT
- What does this mean for you as a programmer?
  - Be aware that your target computer may have limitations (INPUT, PROCESSOR, MEMORY and OUTPUT)
  - Different computers have different programming requirements

INTRODUCTION TO C – FUNDAMENTAL CONCEPTS

- C is a *compiled* language
  - This means a *compiler* takes code from a text file and creates a *binary*
  - The binary can then be *executed* by a computer
      - In this course "computer" will be a lab PC or cloud server
      - Could be a microcontroller, mobile phone, supercomputer cluster, etc
- We will start with http://onlinegdb.com
  - Website contains an editor, code is executed on their server
- Later: an *integrated development environment* (IDE)

INTRODUCTION TO C –FUNDAMENTAL CONCEPTS

- Moving data into and out of a C program
  - *Standard Input*: text characters read from a keyboard
    - `stdin`
    - Could also come from other places, beyond this course
  - *Standard output*: text characters sent to the screen
    - `stdout`
    - Typically printed to a *console*. Other destinations beyond this course
  - File I/O: from or to files stored on a hard disk / USB flash drive / etc
    - Covered in later weeks

INTRODUCTION TO C –FUNDAMENTAL CONCEPTS

- Other input/output methods beyond this course:
  - Microcontroller pins (GPIO – in ENGG1500 and ELEC1710)
  - Embedded systems communication standards, Covered in ELEC2720, ELEC3730, MCHA-something
    - I2C
    - SPI
    - UART
  - TCP/IP networking
  - USB devices
  - Loads of others

STRUCTURE OF A BASIC C PROGRAM

Absolute bare minimum:

```
int main() {
    return 0;
}
```

`main()` is a special *function*. It defines where the program starts *executing*.
The *braces* `{ … }` encompass everything inside `main()`.
When execution reaches return 0; the program stops executing (ie: exits).

STRUCTURE OF A BASIC C PROGRAM

- That's nice, but it does nothing!
  - (ok *fine*, it returns zero to the operating system. *Sheesh.*)
    - (You don't need to know what this means)
- Lets add some more code to make it do more than "nothing":

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

STRUCTURE OF A BASIC C PROGRAM

- `#include <stdio.h>`
    - This "includes" a *header file*
    - `stdio.h` holds the definition of the `printf()` *function*
        - The "definition" is called a *function prototype*. More on this in week 5-ish
- `printf("Hello World!\n");`
    - This line *calls* printf()
    - The stuff between the "…" symbols gets sent to the screen (more on this later)
    - \n means "new line". This isn't done unless you tell it to!

STRUCTURE OF A BASIC C PROGRAM

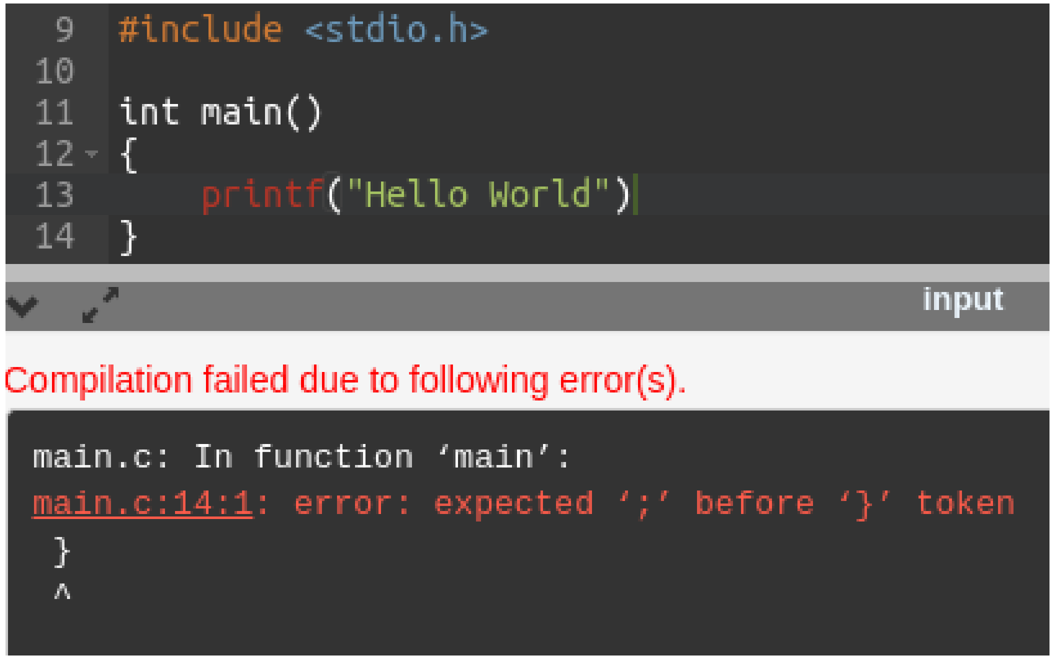
- A few notes:
  - Yes, the code looks weird
    - When was the last time you typed # < > ( ) { } ; ?
  - You need to get used to it
    - This is why the first lab tells you to type out code
  - The `return 0;` is semi-optional.
    - In this course code will run fine without it
    - Compiler may issue a *warning*
  - Likewise: "`int`" before "`main()`" is semi-optional

SYNTAX

- What on Earth is *syntax*?
  - In human languages: the order of words in a sentence
    - Are you going to the movies on Tuesday?
    - Are you on Tuesday to the movies going?
  - In computer languages: the structure of the text given to the compiler
    - For example, the `main()` function syntax is:
      - The word "main" followed by ()
      - An opening brace: {
      - [a bunch of stuff that defines what your program does]
      - A closing brace: }

SYNTAX

- The syntax rules in C are **very** strict
- Missing any syntax rule will result in the compiler generating syntax errors
- Eg:



```
#include <stdio.h>

int main()
{
    printf("Hello World")
}
```

Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:14:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

WHITESPACE

- Whitespace is any tab, space, or newline character
- C *typically* ignores whitespace
  - This formatting is totally fine *for the compiler*:

```
#include<stdio.h>main(){printf("Ohai\n");}
```

  - Humans tend not to like it though

INDENTING STYLES

- Use of whitespace is known as an *indenting style*
  - Examples:

```
int main() {
      printf("stuff");
}
```

```
int
main()
{
      printf("stuff");
}
```

- Adhering to an indenting style makes code easier to read
  - I don't want to be too "religious" about it
  - Just be consistent

CASE SENSITIVITY

- C is *case sensitive*.
- This means that `main()` is **totally different** to `Main()` or `MAIN()`
- Fundamental reason: m and M are different ASCII characters

FUNDAMENTAL CONCEPTS – INPUT

- We saw `printf()` for text output
- One function which reads standard input is `scanf()`
- It reads input text and converts it into other datatypes
  - Eg: the *text* "13" to the *number* 13
- There are many others, keep it simple for now.
- But where does the input go inside the program?
  - New fundamental concept: **variables**

FUNDAMENTAL CONCEPTS - VARIABLES

- A *variable* is something that stores data
  - "Data" is one or more numbers
  - Each variable needs a unique name
  - The compiler decides where in memory it is stored
    - More on this in later weeks – it gets complicated
    - For now: Different variables are stored at different *memory locations*
  - In C variables have a specific *type*
    - For now: lets look at `int`

FUNDAMENTAL CONCEPTS - VARIABLES

- The `int` datatype can store any number from –2147483648 to 2147483647
  - Designed to store integers
    - Can't store "fractional" components. 2.7 = 2.
  - The weird numbers above are:
    - $2^{31}-1$
    - -2^{31}
- Before use, all variables must be *declared*
  - Eg: `int a;` creates a variable of type `int` called "`a`"

`SCANF()` EXAMPLE

- `scanf("%d", &a);`
- Reads an integer and stores it in the variable a
- The "`%d`" is called a *format specifier.*
- `&a` means "the address of a"
  - It tells `scanf()` where to store the number it reads
  - ie: In the memory location allocated to the variable a
- Lots of details glossed over, more on all this later

INPUT-OUTPUT C EXAMPLE

- Lets put together a variable, `scanf()`, and `printf()`:
- What happens if you don't type an integer? Lets run the example...

```
#include <stdio.h>

int main() {
     int a;
     scanf("%d", &a);
     printf("You typed the number %d\n", a);
     return 0;
}
```

FUNDAMENTAL CONCEPT: ASSIGNMENT

- Computer languages use the = character for *assignment*
  - This is **distinctly different** from algebraic equality!
- Assignment means:
  - "Take what's on the right side and store it in the thing on the left"
  - You can read "a = a + 5" as "a **becomes** a plus 5"
- Eg: Give the variable `x` the value 2:
  - `x = 2;`
- Eg: Add `a` and `b` together, store the result in `c`:
  - `c = a + b;`

PUTTING IT ALL TOGETHER

- Lets write a program which:
  - Reads 2 integers from `stdin` on 2 lines
    - "on 2 lines" means the user needs to press enter after each number
  - Multiplies them together
  - Prints the result to `stdout`

SIMPLE PROCESSING EXAMPLE

```c
#include <stdio.h>

int main() {
     int a, b, c;
     scanf("%d", &a);
     scanf("%d", &b);
     c = a*b; // The * character means multiply
     printf("%d\n", c);
}
```

FUNDAMENTAL CONCEPTS: COMMENTS

- In C all text after // is a *comment*
  - Text between /* and */ is also a comment
- A comment is text which is ignored by the compiler
- Use comments to explain your code
  - Code written more than 2 weeks ago might as well had been written by somebody else
- Assume the next person reading your code is a stalker who knows where you live
- "Good" commenting is highly context dependent
  - In this course, assume comments are for a peer who is aiming for 50%.