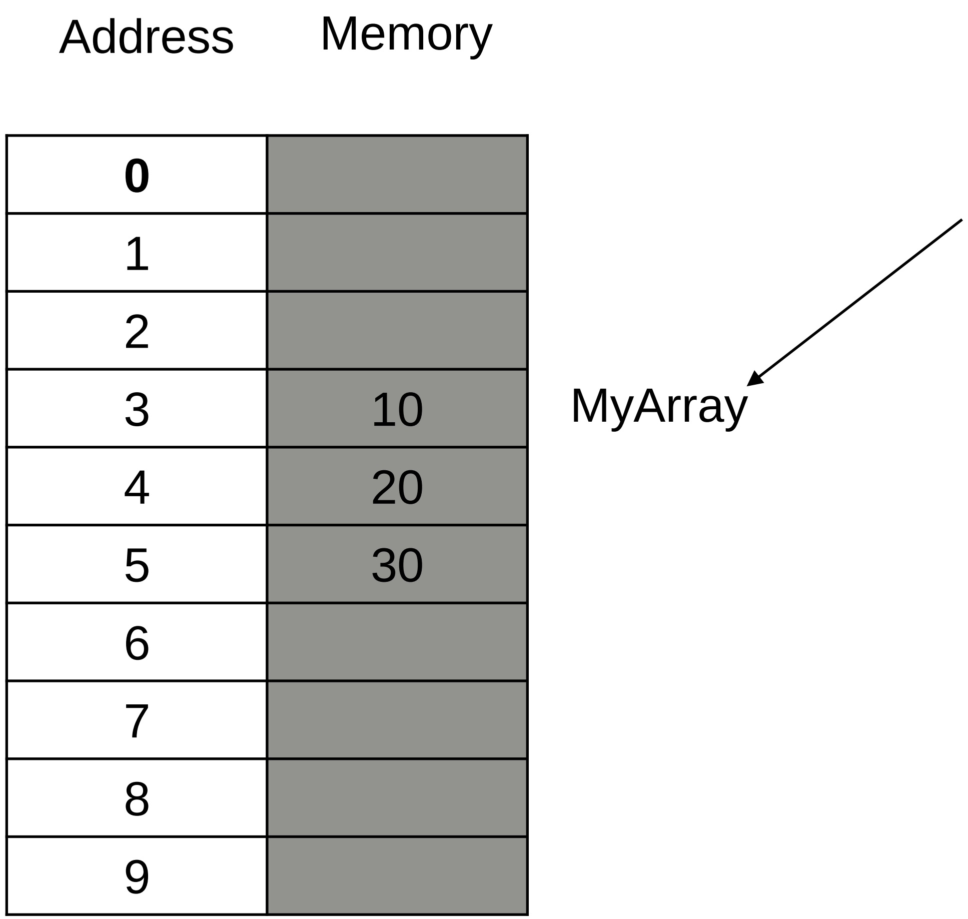
University of Nottingham
UK | CHINA | MALAYSIA

# Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

# Arrays and pointers recap (1)

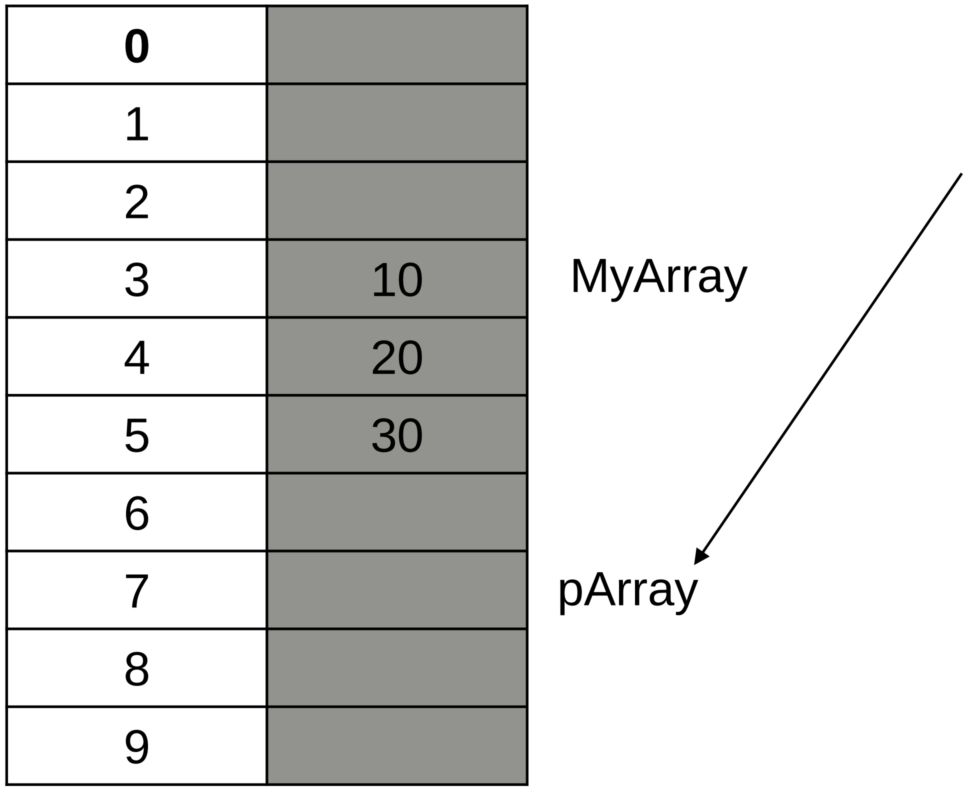


| Address | Memory |
|---|---|
| **0** | |
| 1 | |
| 2 | |
| 3 | 10 |
| 4 | 20 |
| 5 | 30 |
| 6 | |
| 7 | |
| 8 | |
| 9 | |

MyArray

```
char MyArray[3] = {10,20,30};
```

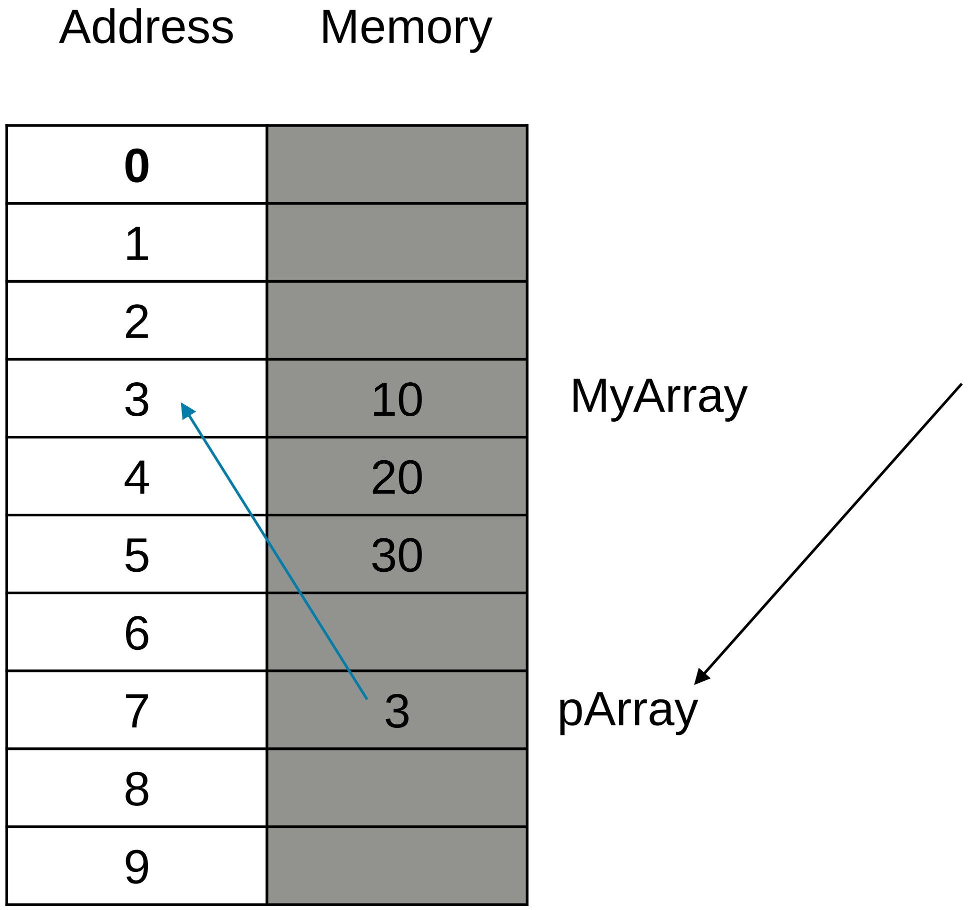
# Arrays and pointers recap (2)



| Address | Memory | |
|---|---|---|
| **0** | | |
| 1 | | |
| 2 | | |
| 3 | 10 | MyArray |
| 4 | 20 | |
| 5 | 30 | |
| 6 | | |
| 7 | | pArray |
| 8 | | |
| 9 | | |

```
char MyArray[3] = {10,20,30};
Char *pArray;
```

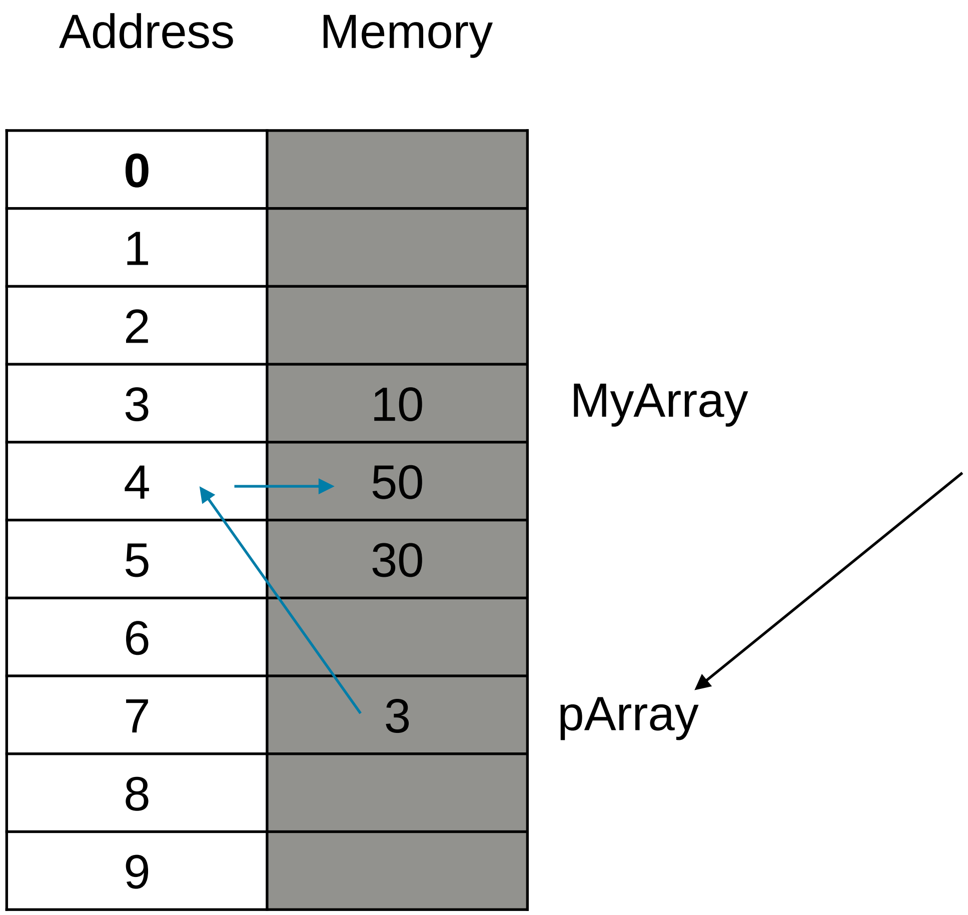
# Arrays and pointers recap (3)



| Address | Memory |
|---|---|
| **0** | |
| 1 | |
| 2 | |
| 3 | 10 |
| 4 | 20 |
| 5 | 30 |
| 6 | |
| 7 | 3 |
| 8 | |
| 9 | |

```
char MyArray[3] = {10,20,30};
Char *pArray;
pArray = MyArray
```

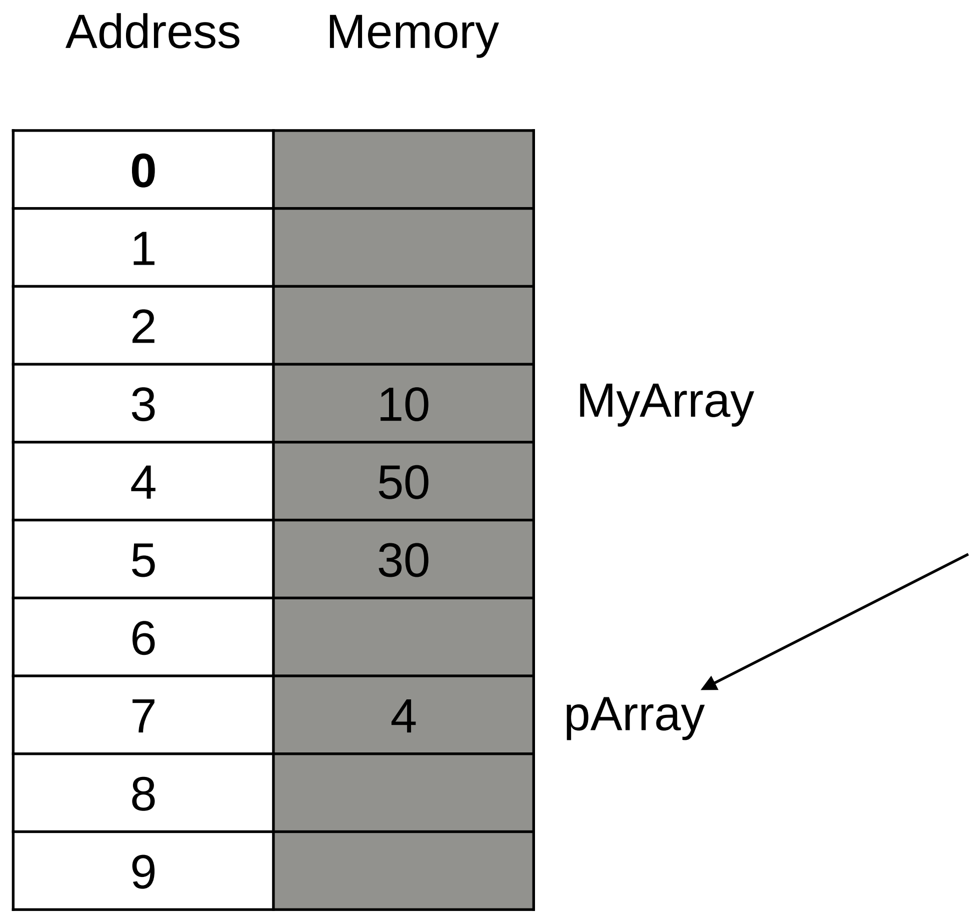
# Arrays and pointers recap (4)



| Address | Memory |
|---|---|
| **0** | |
| 1 | |
| 2 | |
| 3 | 10 |
| 4 | 50 |
| 5 | 30 |
| 6 | |
| 7 | 3 |
| 8 | |
| 9 | |

```
char MyArray[3] = {10,20,30};
Char *pArray;
pArray = MyArray
pArray[1] = 50;
```

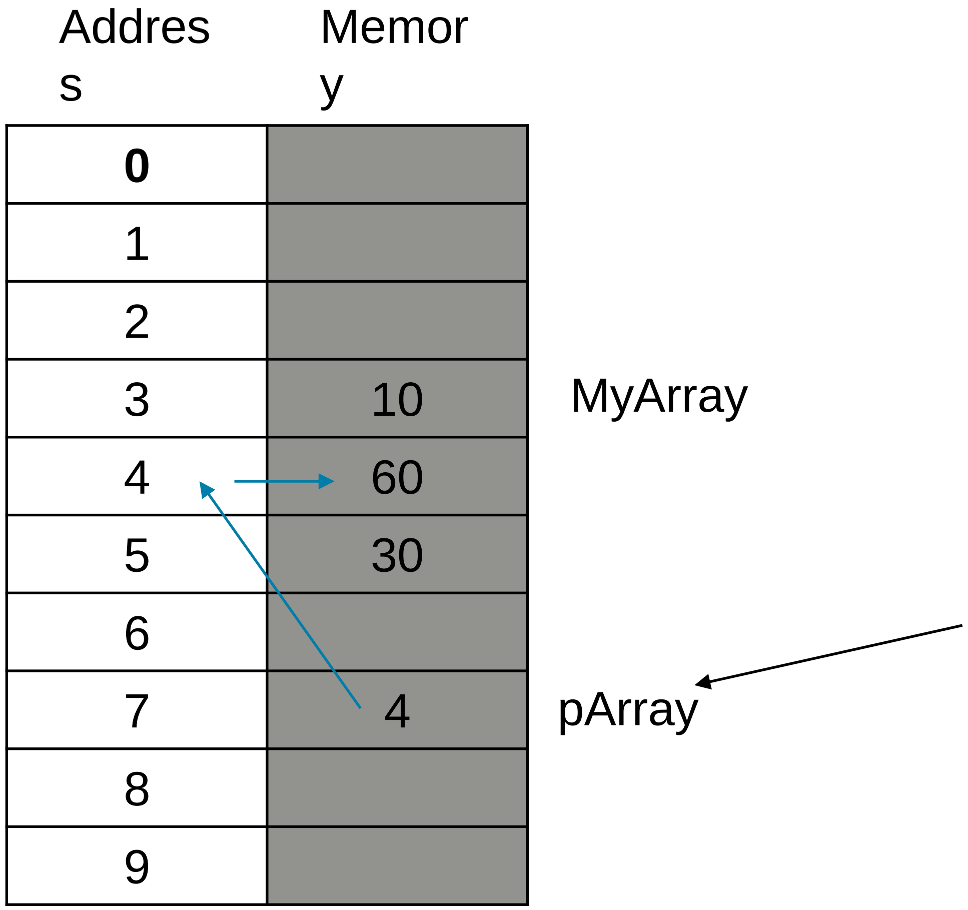
# Arrays and pointers recap (5)



| Address | Memory | |
|---|---|---|
| **0** | | |
| 1 | | |
| 2 | | |
| 3 | 10 | MyArray |
| 4 | 50 | |
| 5 | 30 | |
| 6 | | |
| 7 | 4 | pArray |
| 8 | | |
| 9 | | |

```
char MyArray[3] = {10,20,30};
Char *pArray;
pArray = MyArray
pArray[1] = 50;
pArray++;
```

# Arrays and pointers recap (6)



| Address | Memory | |
|---|---|---|
| **0** | | |
| 1 | | |
| 2 | | |
| 3 | 10 | MyArray |
| 4 | 60 | |
| 5 | 30 | |
| 6 | | |
| 7 | 4 | pArray |
| 8 | | |
| 9 | | |

```
char MyArray[3] = {10,20,30};
Char *pArray;
pArray = MyArray
pArray[1] = 50;
pArray++;
*Parray = 60;
```

# Introduction

Today we will cover:

- Chapter 18 – Using files
- Project introduction
- Software engineering best practice (part 1)

Start recording!!

# Chapter 18

Using Files

# Files

Data stored in a program is *temporary*

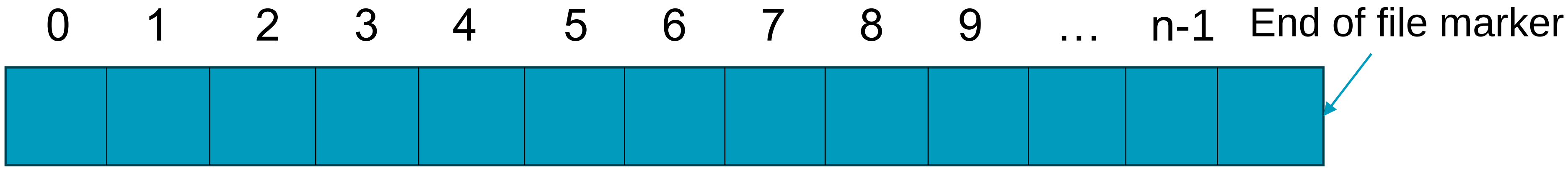
To keep a long-term copy of data it is stored in a file on an external device, eg hard drive, solid state drive, flash drive

# Files

Data stored in a program is *temporary*

To keep a long-term copy of data it is stored in a file on an external device, eg hard drive, solid state drive, flash drive

A C program views a file as a sequential stream of bytes, terminated by an ‘end of file’ marker



Depending on the file type these may be accessed sequentially or randomly

# Text and Binary Files

Text files:

- ✓ Can be viewed in an editor
- ✓ Can be read by different machines
- x Tend to be bulky
- x Must be read in sequence

Binary files:

- ✓ Much Smaller for the same amount of data
- ✓ Can be randomly accessed
- x Can’t be viewed in an editor
- x Byte ordering can be a problem

# Streams

On opening a file, a **stream** is associated with it.

When a program opens 3 streams are automatically created:

- **standard input** which receives data from the keyboard
- **standard output** which displays output to the screen
- **standard error** which displays error messages on the screen

Streams allow communication between a file and the program.

When a data file is opened a stream is created which allows the program to read and write data to and from the file.

# Using Text Files

First, the file must be opened

Once opened, use functions such as

- fscanf and fprintf

In the same way we have been using

- scanf and printf

When finished, it is necessary to close the file

- Not doing so can result in the loss of information!

# Opening Files – fopen (1)

```
FILE *fPtr;

fPtr = fopen( “fred.txt”, “w” );
```

# Opening Files – fopen (2)

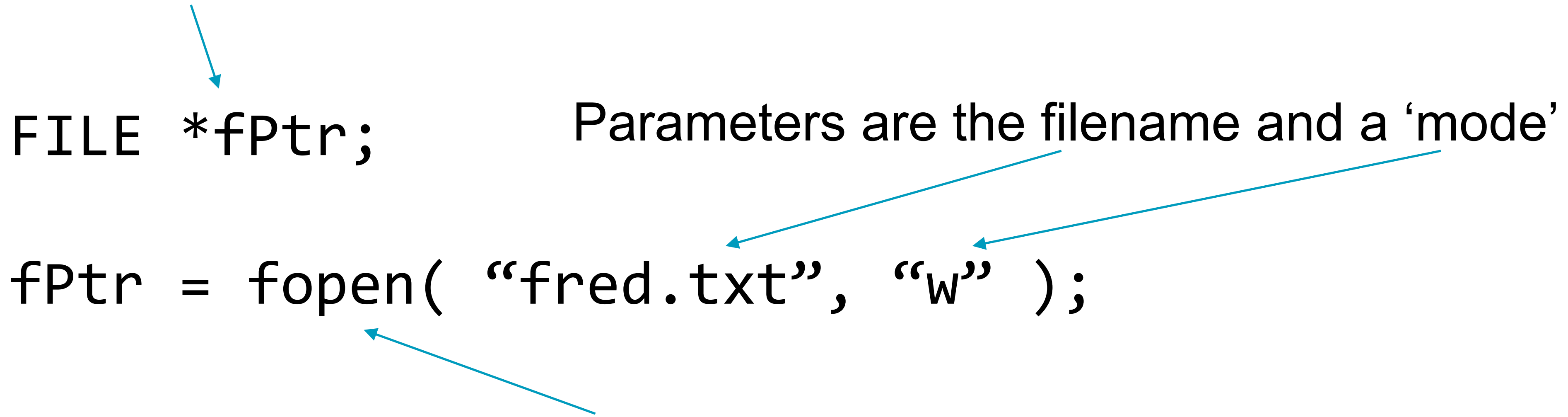
First create a variable of type `FILE *`
This returns a pointer to a FILE structure which is defined in stdio.h

```
FILE *fPtr;

fPtr = fopen( “fred.txt”, “w” );
```

# Opening Files – `fopen` (3)

First create a variable of type `FILE *`
This returns a pointer to a FILE structure which is defined in stdio.h



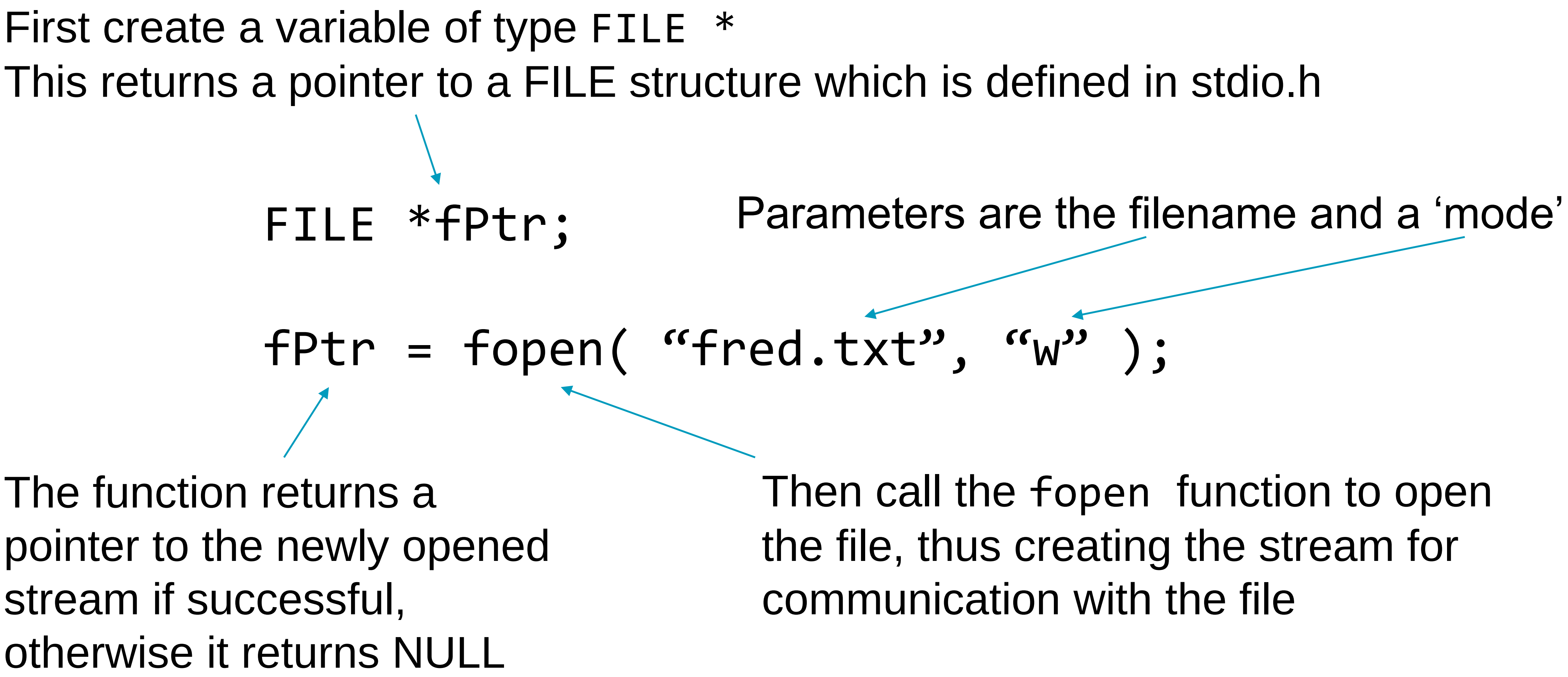
```
FILE *fPtr;

fPtr = fopen( “fred.txt”, “w” );
```

Parameters are the filename and a ‘mode’

Then call the `fopen` function to open the file, thus creating the stream for communication with the file

# Opening Files – `fopen` (4)



First create a variable of type `FILE *`
This returns a pointer to a FILE structure which is defined in stdio.h

```
FILE *fPtr;
```

Parameters are the filename and a ‘mode’

```
fPtr = fopen( “fred.txt”, “w” );
```

The function returns a pointer to the newly opened stream if successful, otherwise it returns NULL

Then call the `fopen` function to open the file, thus creating the stream for communication with the file

# fopen: Text File Examples

Create a new file

```
  fNew = fopen("fred.dat", "w");
```

Open a file for reading

```
  fNew = fopen("fred.dat ", "r");
```

Open a file for appending

```
  fNew = fopen("fred.dat", "a");
```

Note: We use " " here for the ‘mode’ as, though for text files this parameters is (generally) a single character, there are occasions (e.g. binary files) where is it a string

Note: As with memory allocation, we MUST check the file was opened OK!

# Reading and Writing Data To and From Text Files

Functions fscanf and fprintf are used:

Nearly the same as fscanf and fprintf **but** must pass the file handle created when the file was opened:

```
fOutput = fopen("fred.txt", "w");
fprintf( fOutput, “%d\n”, i);

fInput = fopen("fred.txt", “r");
fscanf( fInput, “%d”, &d );
```

# Closing Files - `fclose`

Closes a previously open file

Prototype is in stdio.h

```
fclose( fPtr );
```

Returns:

- 0 : if the file was closed OK
- EOF : if an error was detected

Always call `fclose` as soon as possible when it is no longer needed

- This frees up system resources and leaves the file available to be opened by other programs
- If not explicitly called the operating system should close the file automatically when the program exits (but this cannot be guaranteed)

# Text File Example

We will

- Open ( & create ) a new file,
- Check it was opened OK
- Write the values 1 to 10 to it
- Close the file

Then

- Open it back up
- Check it was opened OK
- Read and display the values
- Close the file

C18\text_file_example.c

# Reading to the end of a file

There will be occasions where we do not know the size of a file so this poses a problem:

- How much data is in the file ?

We could be organized

- Specifying this at the start of file
- This is referred to as a Header
- Note: A header need not be a single value, it can be a complete descriptor of the data contained in the file

If we do not have a header

- We read to the end of file
  - Making use of the function (actually macro) ‘feof’

  or
  - Making use of the return value of functions used for reading data: fscanf, fread, fgetc etc.

# File read return values

Return values:

**`fscanf`** – Returns the number of items of the argument list successfully filled. Returns EOF (end of file ) if this is reached while reading.

**`fread`** – Returns the number of full items successfully read (may be fewer than the number specified if an error occurs). Returns EOF (end of file ) if this is reached while reading.

**`fgetc`** – Returns the character that is read as an integer. Returns EOF if there is an error or the end of file is reached

# Text File Example - modified

Time for an example…

We will

- Open ( & create ) a new file,
- Check it was opened OK
- Write the values 1 to 10 to it
- Close the file

Then

- Open it back up
- Check it was opened OK
- Read and display the values using the return value from **fscanf** to read to the end
- Close the file

LC18/ EndOfFileScanf

# Using the `feof` macro

Macro that checks for the End Of File

- Returns non-zero if end of file reached
- Returns zero otherwise
- Works for text and binary files

Prototype is in stdio.h

```
  int feof ( FILE *handle )
```

Be careful – Checks the current state of the file handle

We will look at the same example but using **feof** to find the end of file

# Binary Files

A binary file is one where the data written to the file is the bytes used for storage, rather than the ‘text’ format

We are, in effect, copying areas of memory to and from the file

# The Advantages

The files use less space:

For an integer, value 32768:

- Binary file – 4 bytes
- Text file – 5 characters (10 bytes)

For a float, value 78935670394587289473659.893475629836745:

- Binary file – 8 bytes
- Text file – 39 characters (78 bytes)

# More Advantages

We can write whole arrays/structures in one go

- This is much faster than having to convert each value to its text format and output it

Also, each item is always the same size

- We will make use of this property later

We cannot directly read/edit/print binary files

We can hit a problem with byte ordering (and storage size) when going across platform

- e.g. SUN to/from PC

# Opening a Binary File

Basically the same as a text file, We just add an extra bit to the mode to indicate it is a binary file, e.g.

Create a new file

```
    fNew = fopen( "fred.dat", "wb");
```

Open a file for reading

```
    fNew = fopen( "fred.dat", "rb");
```

Open a file for appending

```
    fNew = fopen( "fred.dat", "ab");
```

Note: As with memory allocation, we MUST check the file was opened OK!

# Using the File

There are a few ‘common’ functions we use for reading/writing to binary files

- `fread`
- `fwrite`
- `fgetc`
- `Fput`

There are others but we do not need worry about these - see the help system for more info!

When we have finished, we (as with text file) close the file using

- `fclose`

# fgetc, fputc

These work the same as getchar and putchar, allowing us to read or write single characters to a file - we just include a pointer to the file

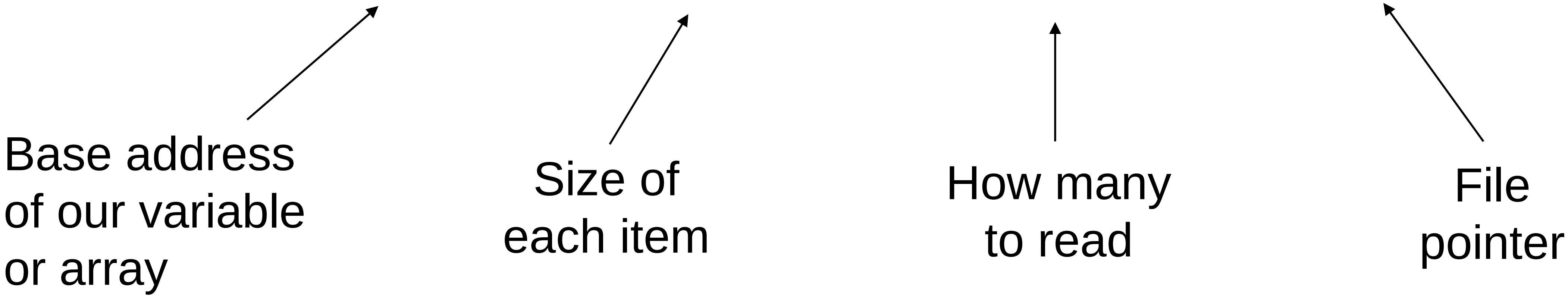
```
int fgetc( FILE *stream);
int fputc( int c, FILE *stream);
```

# The ‘new’ commands: fread

**fread**

Reads (copies) binary data from a file directly into memory, starting at the memory location we specify

```
#include <stdio.h>
size_t fread  (void *ptr,   size_t s,   size_t n,  FILE *stream);
```



Base address of our variable or array

Size of each item

How many to read

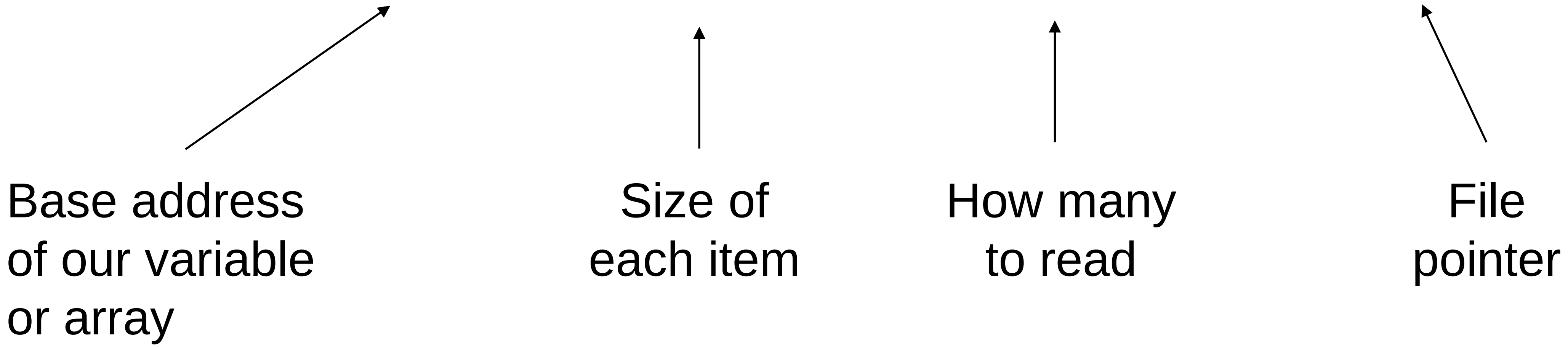
File pointer

# The ‘new’ commands: fwrite

**fwrite**

Writes (copies) a block of memory, starting at the memory location we specify, to a binary file

```
#include <stdio.h>
size_t write  (void *ptr,   size_t s,   size_t n,  FILE *stream);
```



Base address of our variable or array

Size of each item

How many to read

File pointer

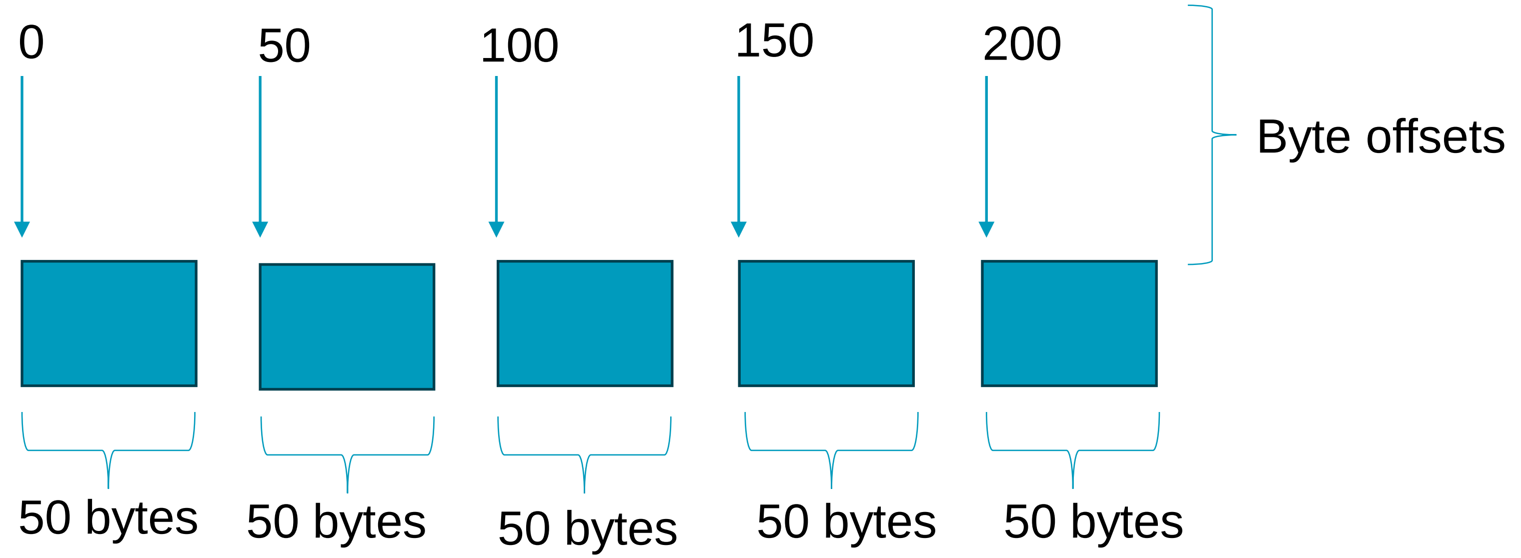
# Time for an example

We will

- Define an array of 10 elements
- Populate the array
- Write it out to file (in one go!)
- Read it in again

C18\binary_file_example.c

# Random Access in Binary Files



Records in binary files are typically all the same (known) size. This makes is easy to calculate the location of a specific entry, making it possible to move straight to that entry to read or write data.

We can ‘jump’ around the file using the `fseek` function

# fseek

The **fseek** function is defined in stdio.h:

```
int fseek(FILE *stream, long offset,  int whence);
```

File pointer

How far to move

Where to move from

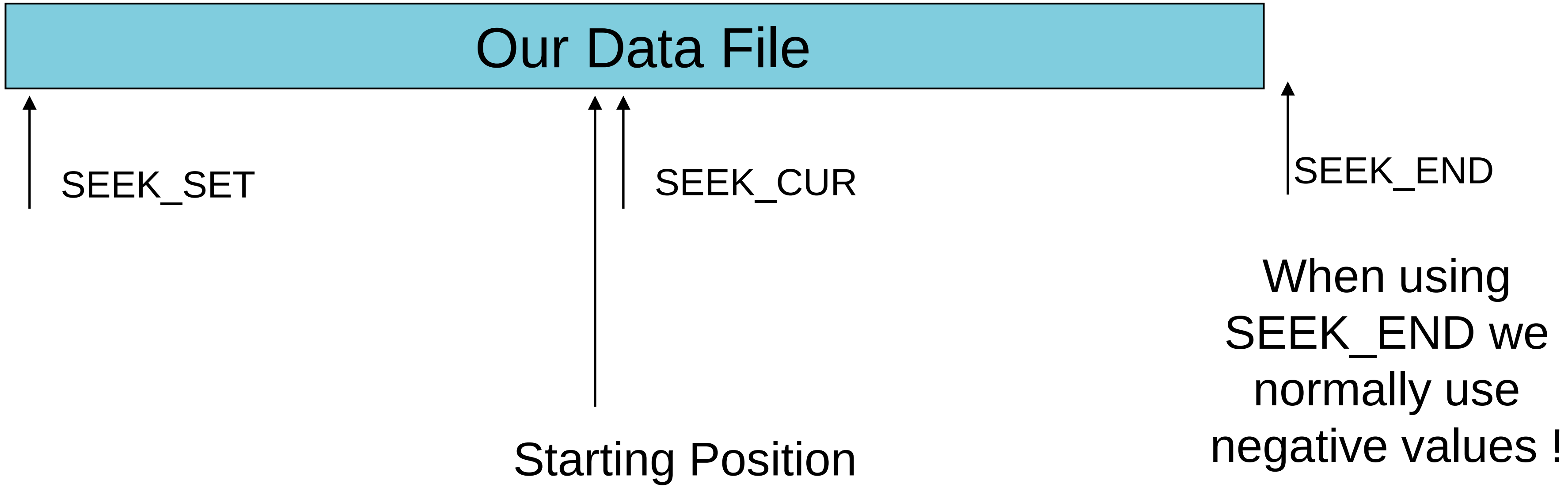
If whence is ‘SEEK_SET’  : The move is made from the start of the file

If whence is ‘SEEK_CUR’ : The move is made from the current position

If whence is ‘SEEK_END’ : The move is made from the end of the file

# Graphically

```
fseek ( fptr , 1, whence )
```



When using SEEK_END we normally use negative values !

Another useful command:

`rewind (fptr)`

It is the same as `fseek( fptr, 0 , SEEK_SET)`

C18\filemove.c

# Another example

We will ask the user which value they wish to view

- We move forward to this value
- We display the value on the screen

LC18\filemove.c

# How big is my file ?

Unlike text files we do not need to read to the end of the file to determine the number of items contained within, rather:

- We ‘seek’ to the end of the file using the SEEK_END parameter
- We then get the current file position in bytes (size of the file)

```
long ftell ( FILE *stream )
```

As we know the size of each element:

number of items = size of file / size of element

LC18\filesize.c

# Structures: A ‘neat’ way to write headers

A better way however is, as with text files, to have a header at the start

We often make use of a structures as file headers

If we know all about the data to be written to a **binary file** we can

- Write out the header
- Write the data to the file

If, however we are writing data ‘on the fly’ to a **binary file** we can

- Write out a dummy header
- Write the data to the file
- Rewind back to the start of the file
- Write out an updated header with the correct information
    - As the structure remains the same size (even if the contents changes) it is a simple ‘overwrite’

# Using a structure for a file header

We can still ‘randomly access’ the file (to pick specific items) however we need to remember to add the offset caused by the header

e.g.

If the structure is called ‘MyHeader’
All the data are integers

To get to the 5th item we need (i.e. skip 4)

```
Offset = sizeof (struct MyHeader) + 4*sizeof( int )
```

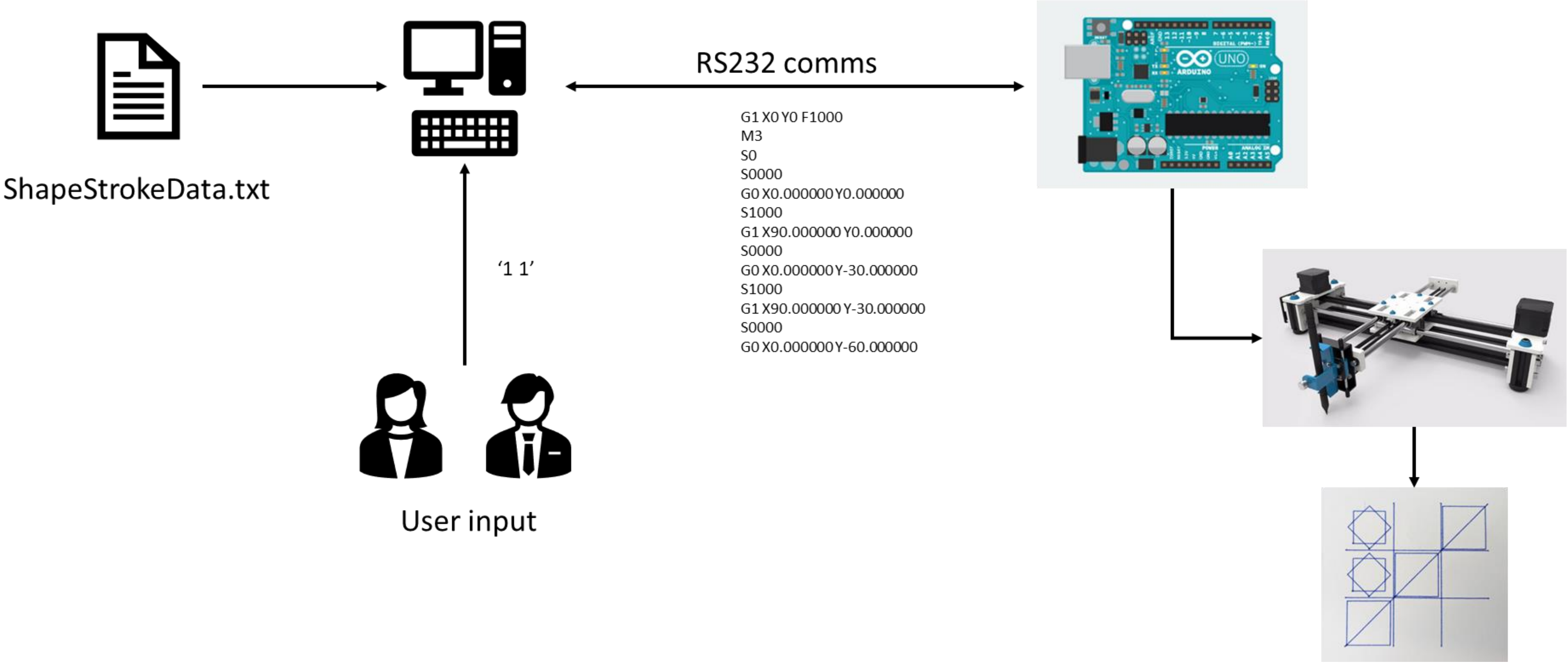
LC19\file_header_move.c

# Software Project

Developing Software for a Drawing Robot

# Overview

See Computer Engineering and Mechatronics Project v6.0.pdf in the Software Project 2023/24 section on Moodle



ShapeStrokeData.txt

RS232 comms

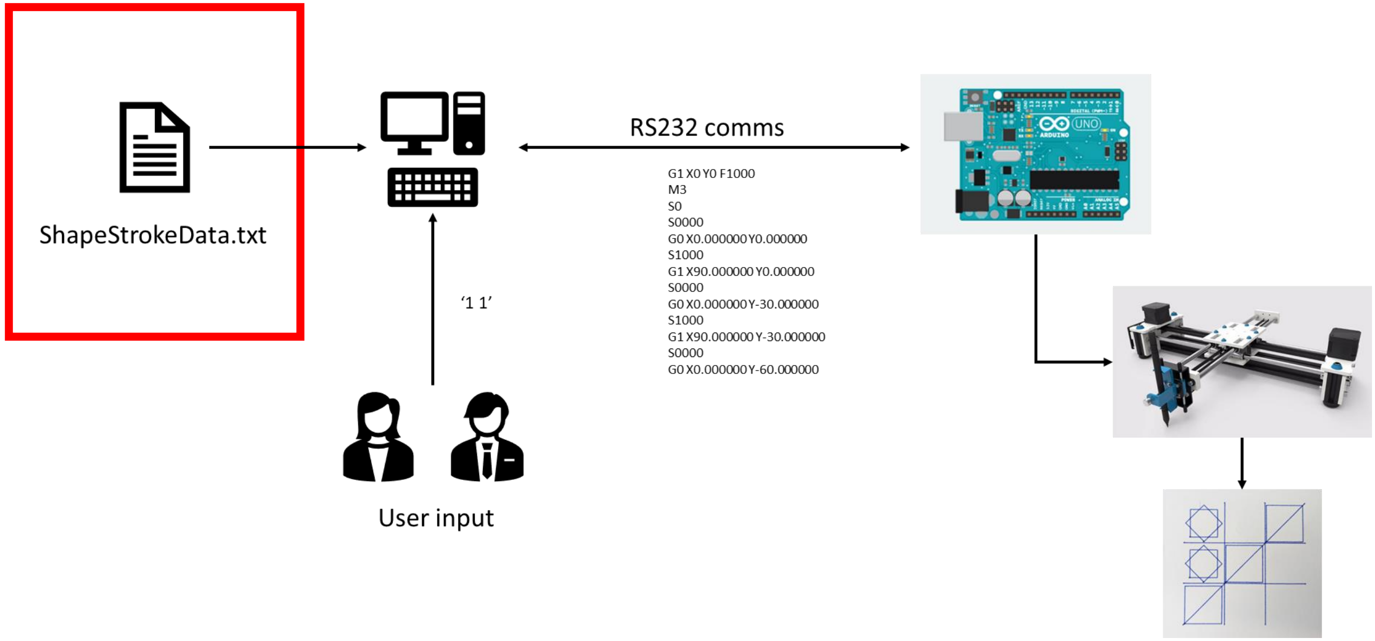
‘1 1’

User input

```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

# Shape Strokes File – Appendix 1

Details of file format for ShapeStrokeData.txt are given in Appendix 1



```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

General format:

```
NumShapes N
SHAPE_NAME S
X Y P
X Y P
X Y P
```

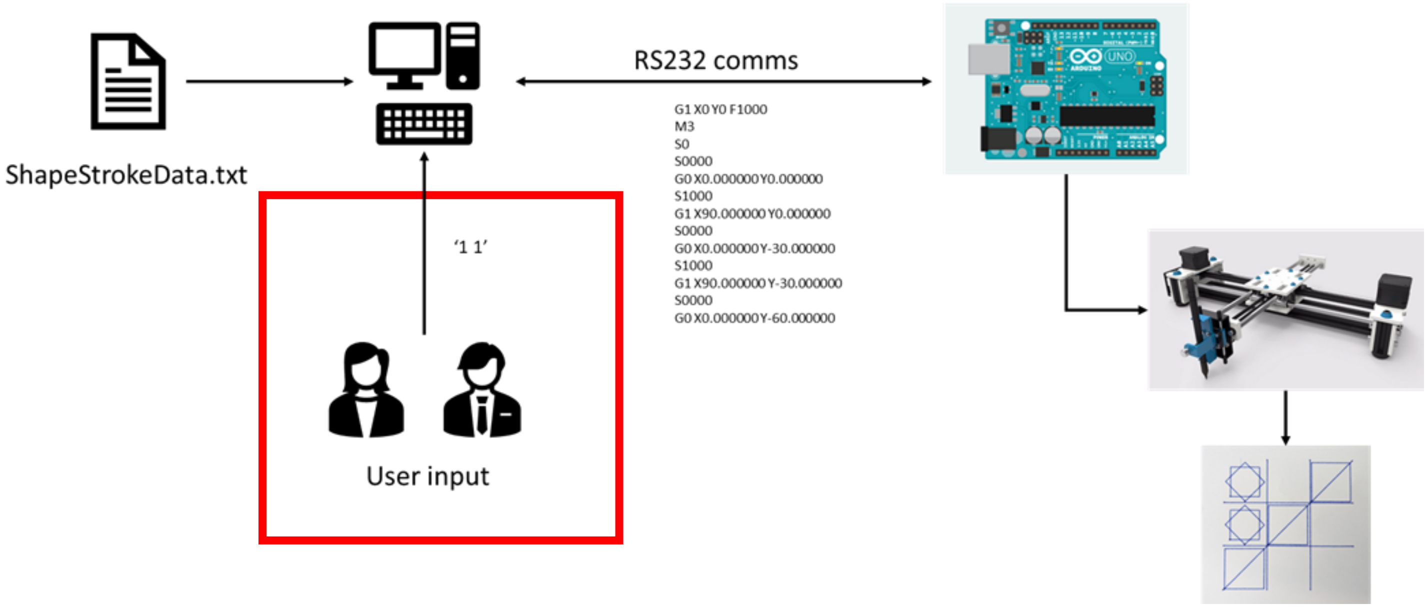
Where:

- NumShapes: Static text (always 'NumShapes')
- N: The number of shapes defined in the file
- SHAPE_NAME: The string identifier for the shape name
- X: The X position to move to (relative to 0,0)
- Y: The X position to move to (relative to 0,0)
- P: Pen up/down (0=up so no line is draw, 1=down so causing a line to be drawn)

# User input and playing the tic tac toe game

User input:

- Select shape to play with
- Input grid size (30 – 100mm)
- Take turns to select the grid position for the move



```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

# G-Code – Appendix 2

You will need to generate G-codes for the text to be written by the robot

The codes will be generated using the shape data read from ShapeStrokeData.txt for the shapes selected by the players at the positions they have chosen for their move in the game. This needs to be scaled to give the correct size.

Use the subset of G-codes shown here:

| Command | Description |
|---|---|
| F1000 | feed rate (i.e. pen speed) 1000 mm min$^{-1}$ |
| G0 X Y | Move to the position X,Y |
| G1 X Y | Draw a straight line from the last position to X,Y |
| M3 | Turn on Spindle (needed for arm to work!) |
| S0 | Pen up (original meaning is ‘spindle speed 0’) |
| S1000 | Pen down (original meaning is ‘spindle speed 1000 rev min$^{-1}$’) |

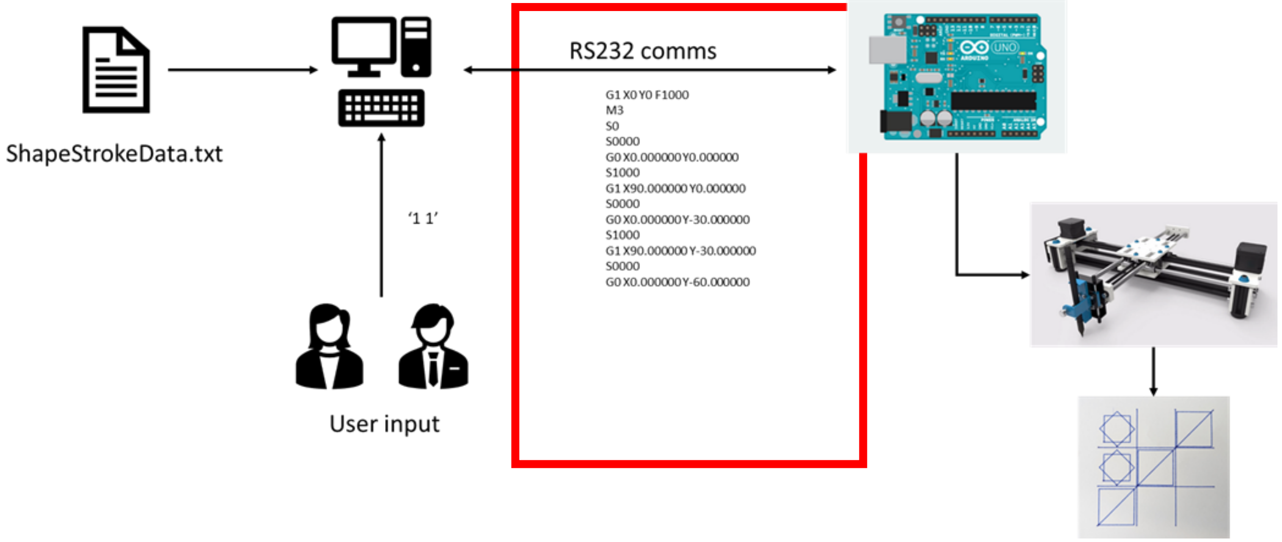
The G-Code Simulator can be used to check if G-codes have been generated correctly

# Serial Communication – Appendix 4

A virtual serial port is used to send the G-Code commands

An RS-232 library written by Teunis van Beelen is used.

This library has been incorporated into an example program, BlinkSerial. Download this from Moodle and follow the instructions to see how this works.

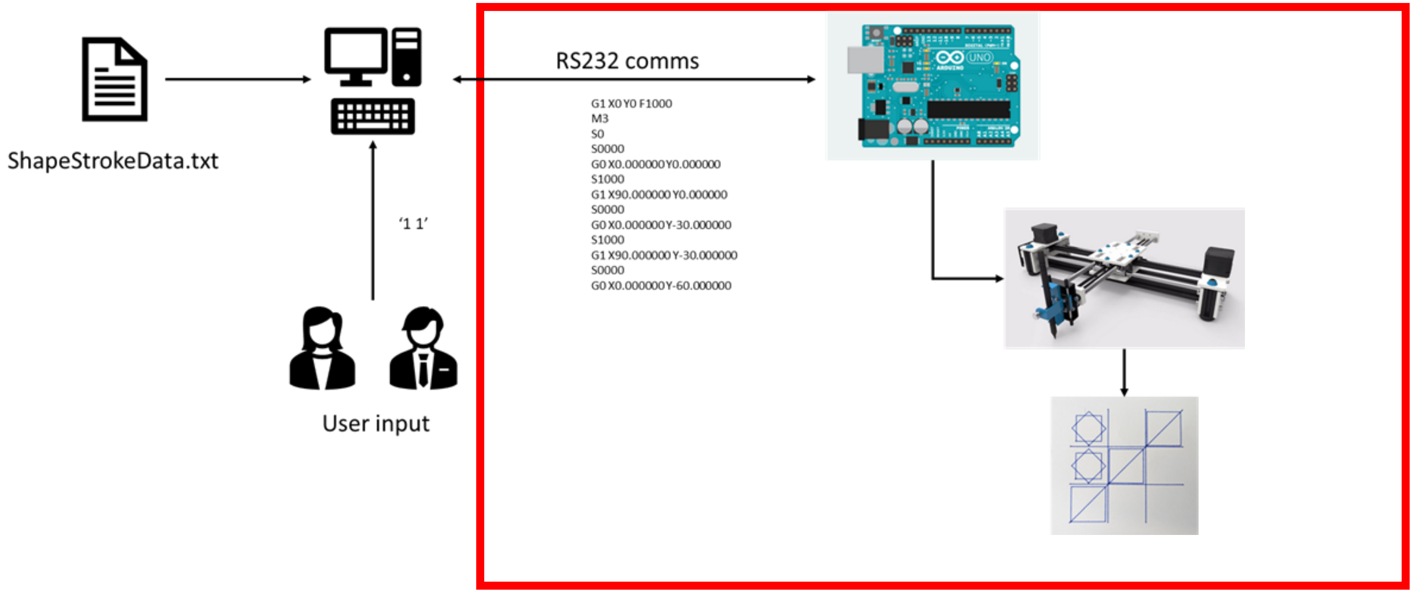


```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

# Sending G-code to Arduino and awaiting acknowledgement – Appendix 4

The sample code in RobotWriter5.0.zip on Moodle gives a sample project for sending some hard-coded G-code

The Serial.c file uses a #ifdef statement to either send the G-code to the serial port or to be printed (to enable testing using the emulator)

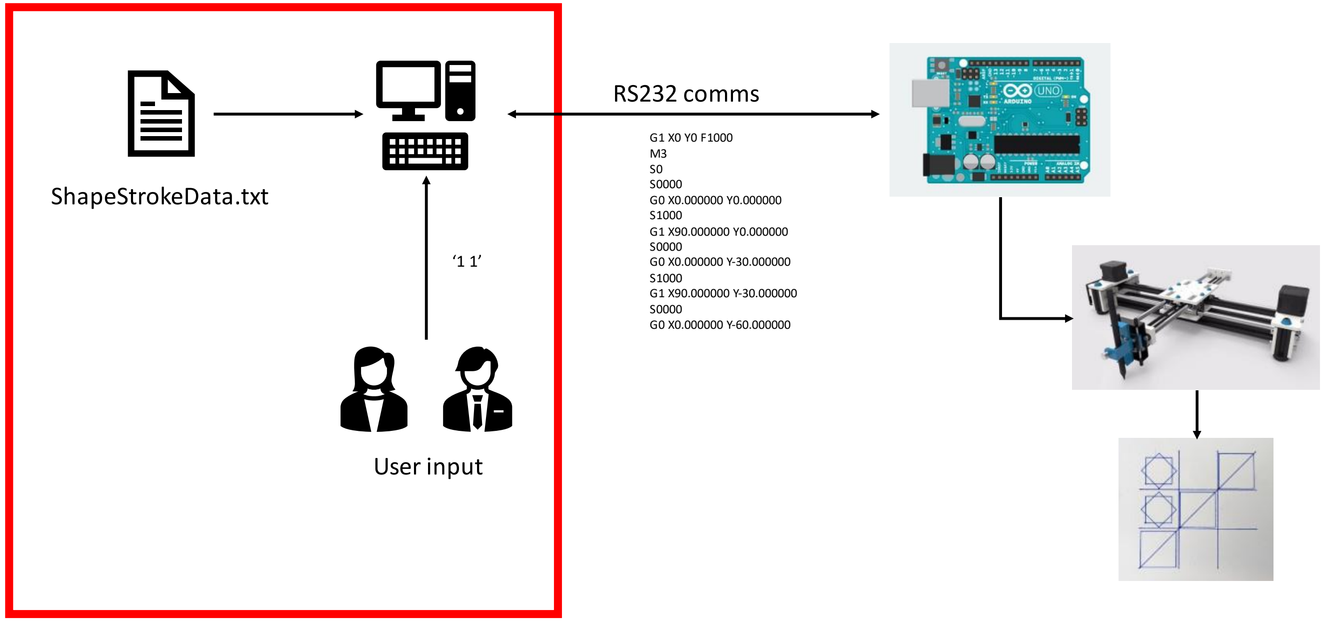


```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

# Software Project

Start with the project in RobotWriter5.0.zip

Replace the hard-coded G-code commands with code which reads the shapes file and generates the G-codes to draw the shapes using the robot



```
G1 X0 Y0 F1000
M3
S0
S0000
G0 X0.000000 Y0.000000
S1000
G1 X90.000000 Y0.000000
S0000
G0 X0.000000 Y-30.000000
S1000
G1 X90.000000 Y-30.000000
S0000
G0 X0.000000 Y-60.000000
```

There is sample code which may help you in the Coding Samples and Examples section on Moodle

There are a set of exercises for the week 6/7 computer lab which will help you to think about how to load and store the shape file. This will be very useful for the project planning.

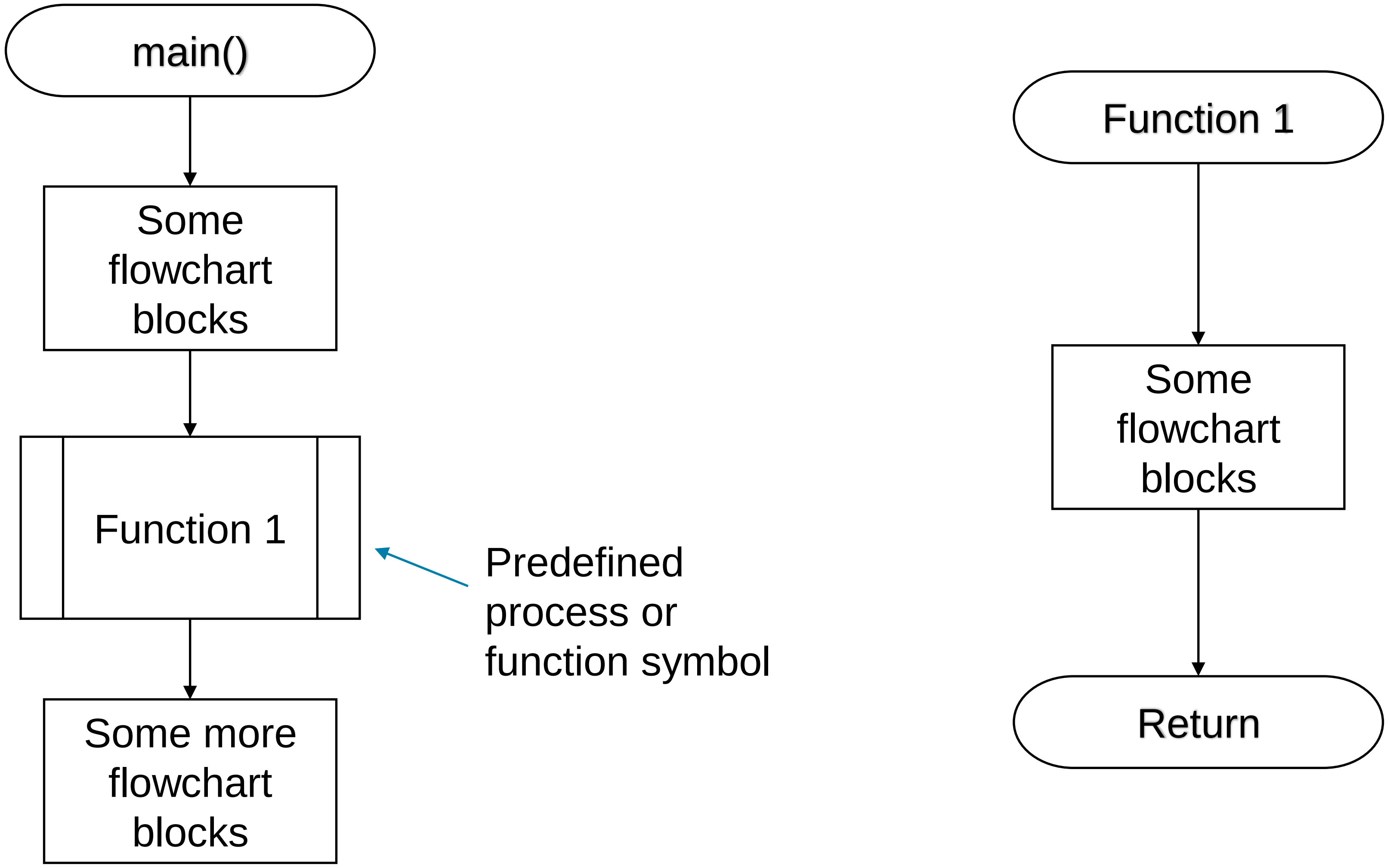
# Submissions: Design Document – 3pm Tuesday 21st November

In line with the design processes covered in the course you will be required to produce a specification document using the template on Moodle (ProjectPlanningTemplate23-24.docx):

- A specification of precisely what the program needs to do
- The forms of the data stored within your program
- The planned function declarations (prototypes) for each function identifying whether parameters are input, output or changed, and the return value if any. You are encouraged to give a return value which indicates successful execution or failure.
- Test cases for each function to confirm conformance of the function to its specification.

You need also to provide a flowchart showing the operational flow of your code.

# Flowchart – predefined process or function



Predefined process or function symbol

# Software Engineering Best Practice

Part 1

# What is Software Engineering?

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software"— IEEE *Standard Glossary of Software Engineering Terminology*

Covers the whole process required to produce a software product

# 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?

These were your ideas at the start of the module – any changes?

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

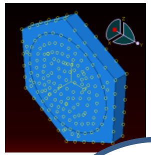
Testing

Deployment

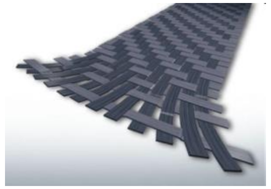
Maintenance

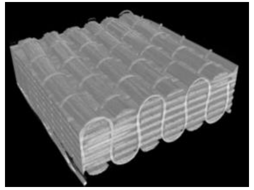
# TexGen Geometric Textile Modelling Software

## Fibre/Micro-Scale

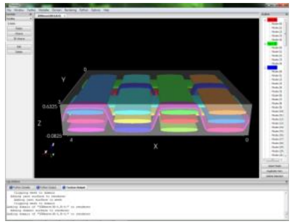


Micro-scale FEA simulations or analytical methods determine yarn properties

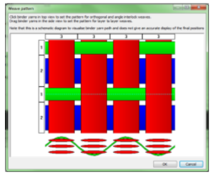




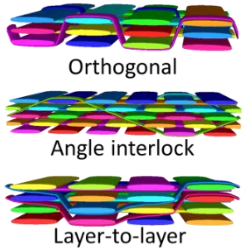
## Unit Cell/Meso-Scale

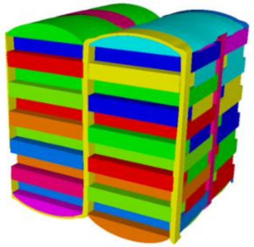


Generate textile geometry using TexGen GUI or script

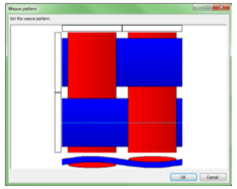


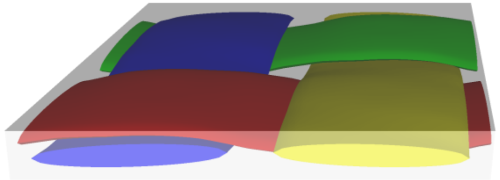
3D wizard generates idealised 3D textiles



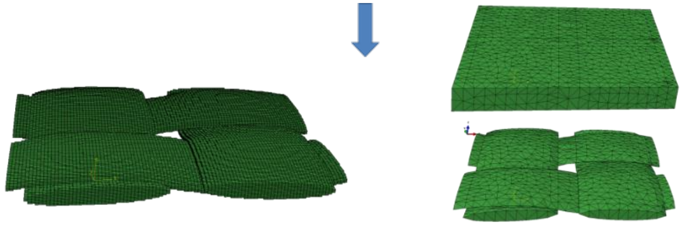


Refinement of orthogonal weave to simulate compaction





Automatically generate 2D and 2D sheared textiles



Generate mesh and input files for FEA or CFD to predict material properties

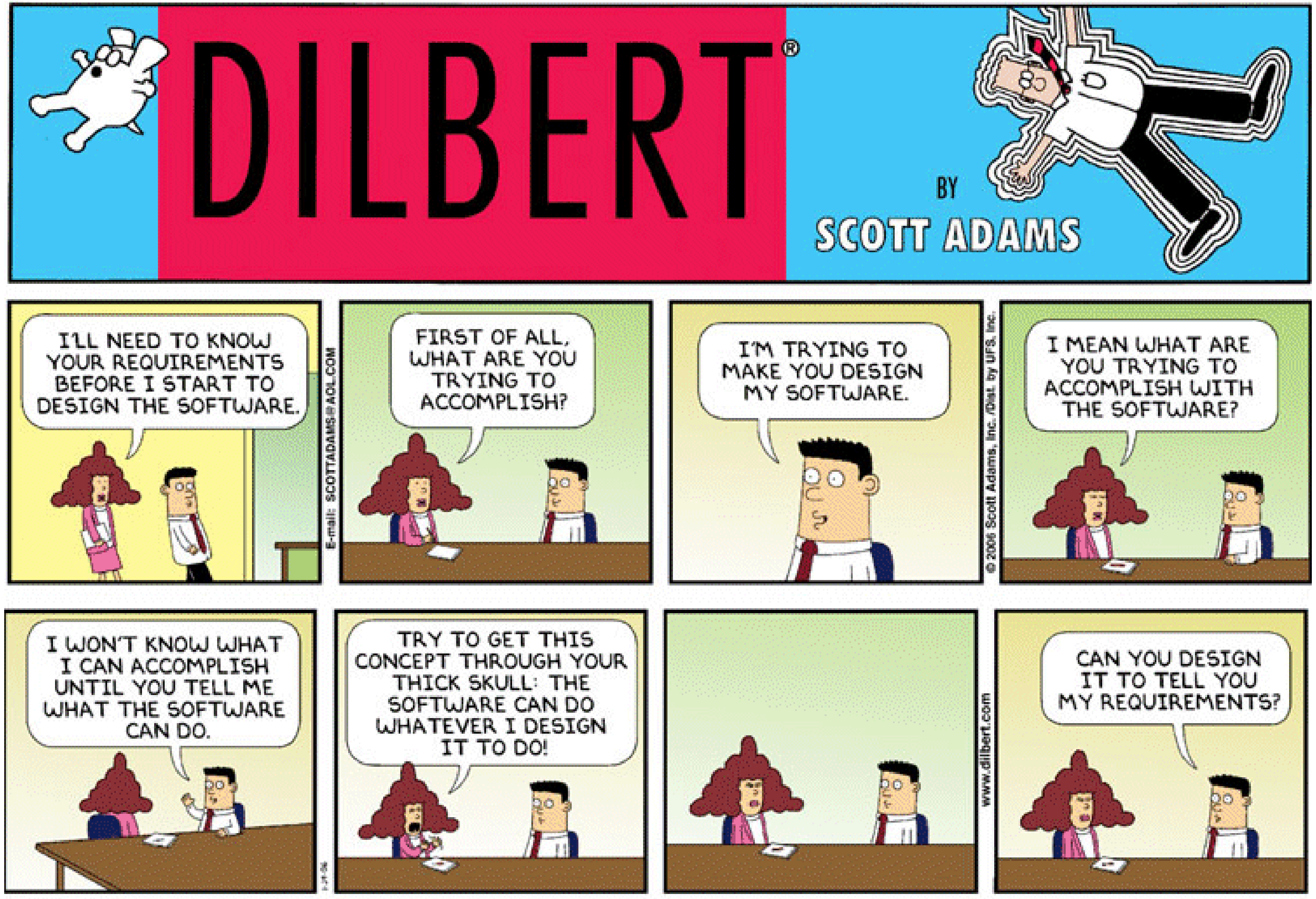
## Component/Macro-Scale

Composite material properties extracted from meso-scale predictions are used to model structural components

# Requirements Gathering

Dilbert

by Scott Adams


DILBERT®
BY SCOTT ADAMS
I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
E-mail: SCOTTADAMS@AOL.COM
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
© 2006 Scott Adams, Inc. /Dist. by UFS, Inc.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
www.dilbert.com
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?

© Scott Adams, Inc./Dist. by UFS, Inc.

# Requirements Gathering

Good design starts with being able to define your problem (in language your user can understand)

*If you can’t explain something to a six-year-old, you really don’t understand it yourself – Albert Einstein*

Specify the *requirements* – what features your software must provide

Must be precise, clear and unambiguous

Prioritise – what are the essentials and which are ‘nice to have’

Verifiable – can it be tested that the requirement has been met?

# High Level Design

Gives an overall view of a system

Defines the major components of a system and their interactions. These can be thought of as a set of building blocks each with its own set of responsibilities. Communication rules between blocks should be well defined.

Specify major classes and data. Think about why a specific data format or file type is to be used. Consider any libraries which can be used.

User interface design. This should not affect the classes and data already specified.

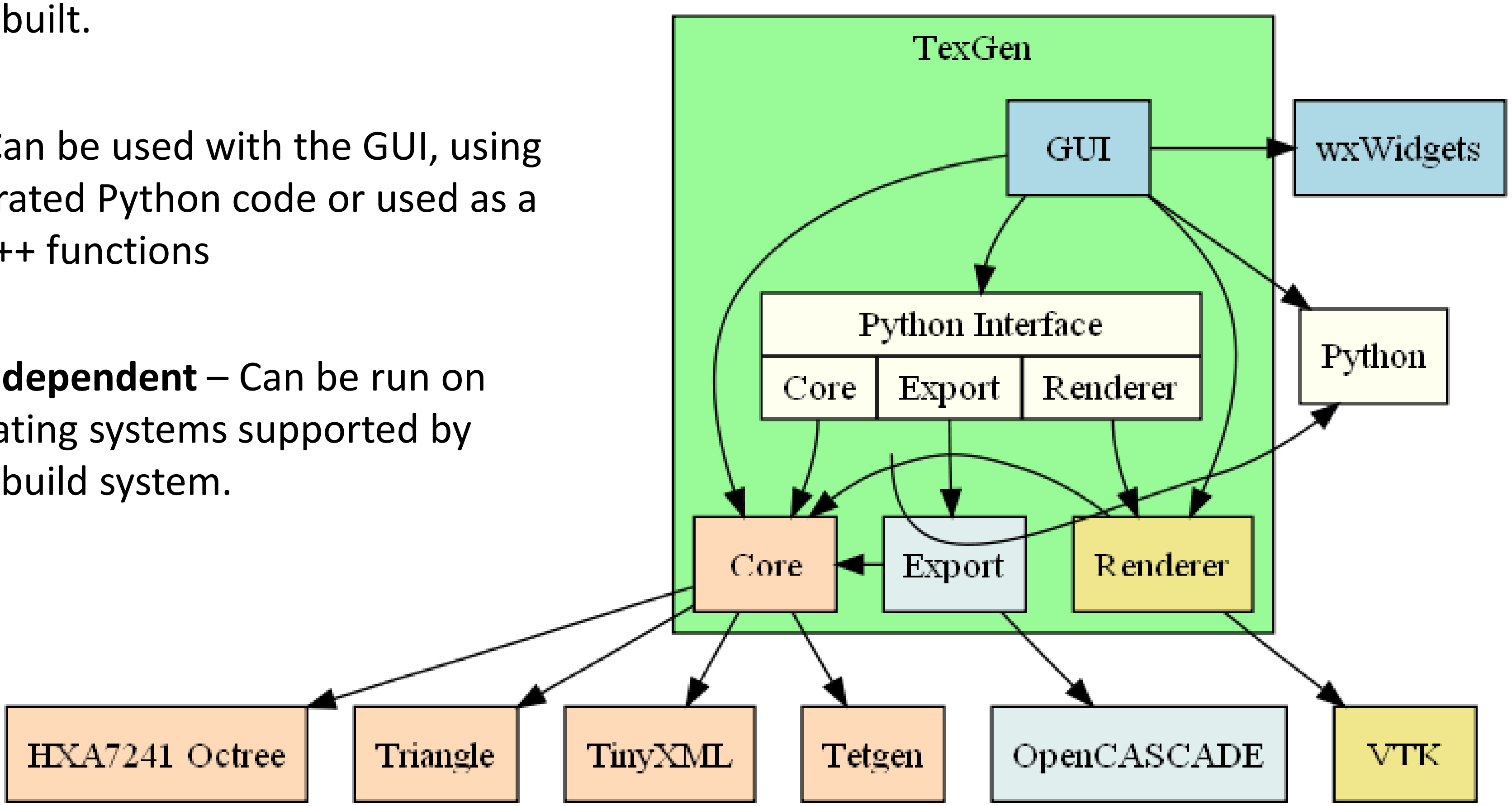
May use tools such as UML (Unified Modelling Language)

# High Level Design

**Modular -**Core functionality is in the core module, graphics are in a renderer module; if not using visualisation, the renderer doesn’t need to be built.

**Flexible –** Can be used with the GUI, using SWIG generated Python code or used as a library of C++ functions

**Platform independent** – Can be run on most operating systems supported by the CMake build system.



# Low Level Design

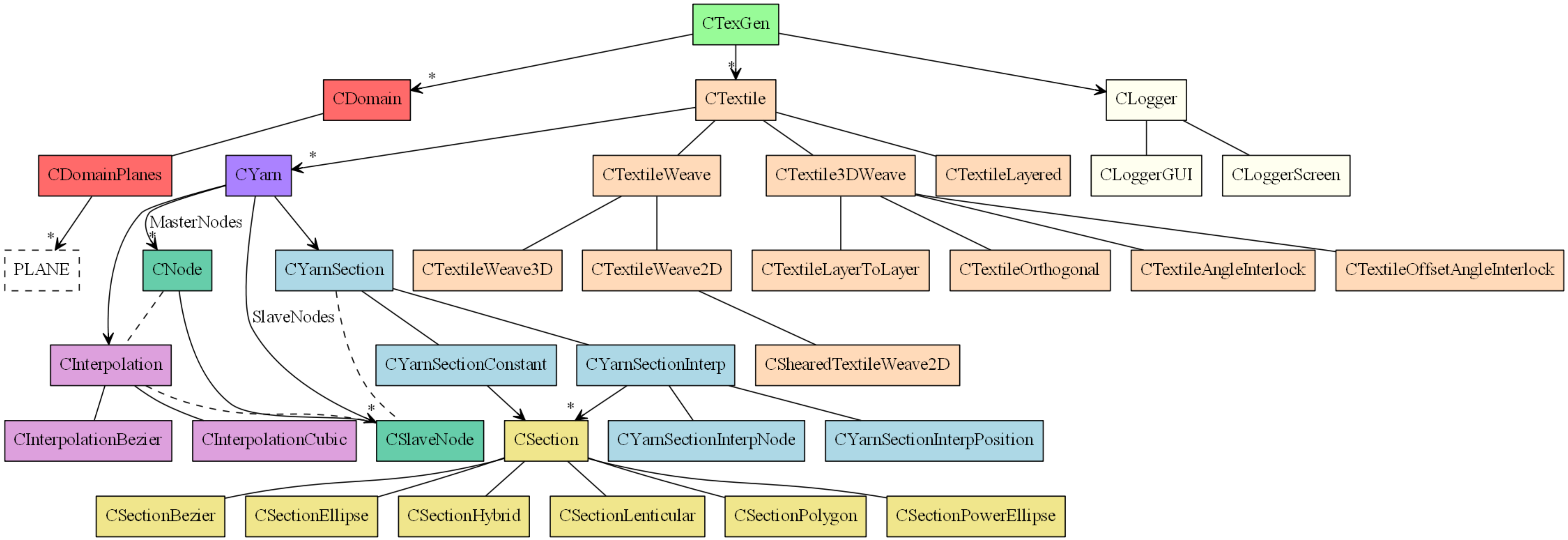
Provides the detail about how the high level design will be implemented.

Don’t dive into the detail straight away. Start to refine the detail of what functions will do, what classes or data structures are needed.

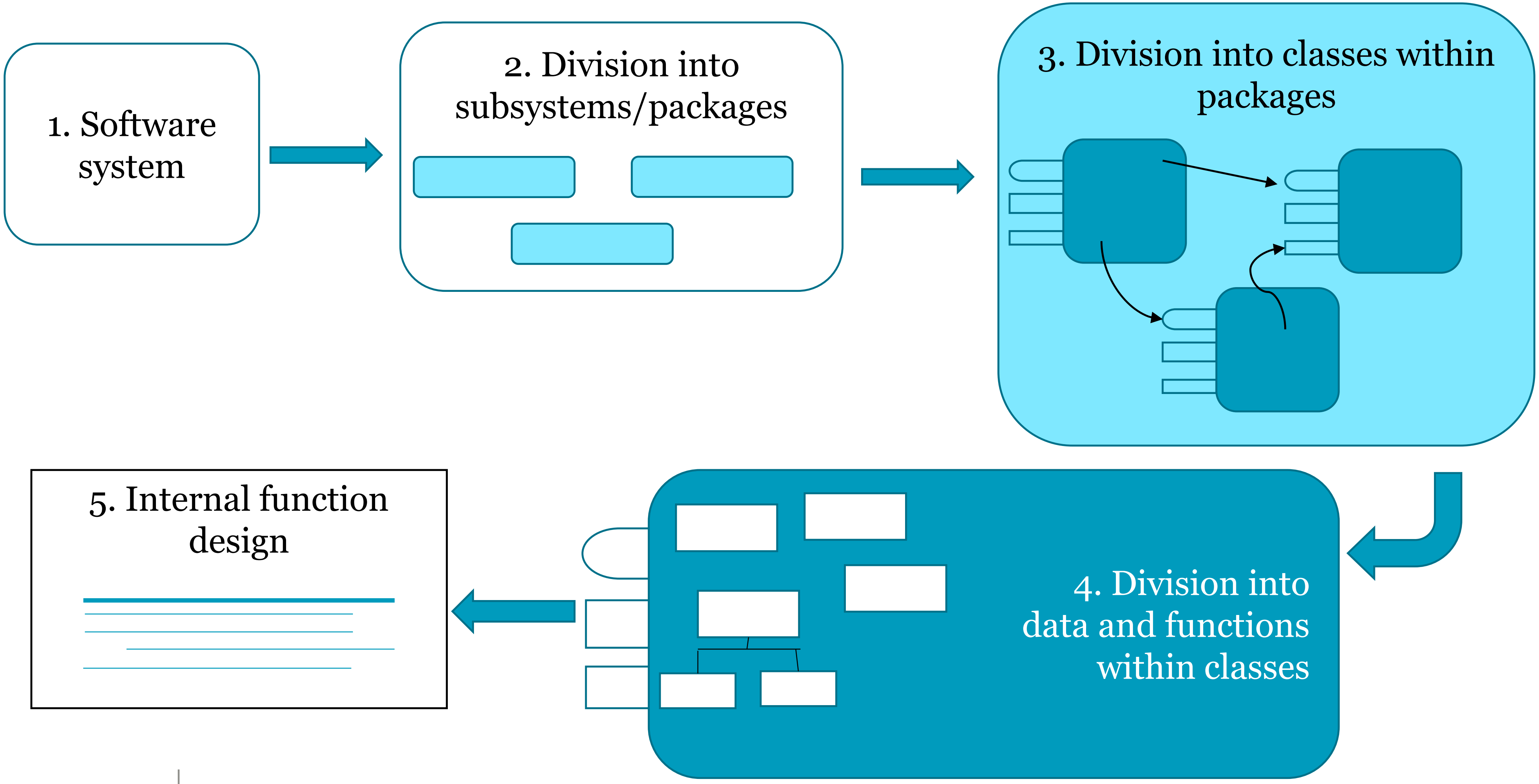
Define the interface – what is passed in and out of a function, what parameters can be changed

This can be an iterative process. For example if several functions all pass the same set of parameters it may be that these should be grouped together in a structure so the data structure may need to be revisited.

# TexGen Core Class Heirarchy



# Levels of Design



1. Software system

2. Division into subsystems/packages

3. Division into classes within packages

4. Division into data and functions within classes

5. Internal function design

# Robot Writing Project

1. Design Document:

In line with the design processes covered in the course you will be required to produce a two page specification document which provides

- An explanation of precisely what the program needs to do
- The forms of the data stored within your program
- The planned function declarations (prototypes) for each function identifying whether parameters are input, output or changed, and the return value if any. You are encouraged to give a return value which indicates successful execution or failure.
- Test cases for each function to confirm conformance of the function to its specification.
- Flowchart(s) showing the operational flow of your code.

# Chapter 19

Advanced Data Types in C – Advanced Structures

# Advanced Structures

A C struct can have bit fields

- append a : and a number to an integer type

```
struct SmallNumbers
{
   unsigned int a:4;
   unsigned int b:4;
   unsigned int c:4;
   unsigned int d:4;
};
```

# Bit Fields

```
struct SmallNumbers
{
   unsigned int a:4;
   unsigned int b:4;
   unsigned int c:4;
   unsigned int d:4;
};
```

`struct SmallNumbers` has 4 members

- Each member has 4 bits
- The value each can take is defined by the number of bits
- The structure is automatically made the correct size
- Structure parts are independent of each other

# Another example of bitfields

```
struct Bits
{
   unsigned char  b0 : 1;
   unsigned char  b1 : 1;
   unsigned char  b2 : 1;
   unsigned char  b3 : 1;
   unsigned char  b4 : 1;
   unsigned char  b5 : 1;
   unsigned char  b6 : 1;
   unsigned char  b7 : 1;
};
```

Assigning: `struct Bits cByte = {0,1,1,0,1,1,1,1};`

Or `cByte.b0 = 0;`

```
   cByte.b1 = 1;
```

# We can also leave gaps

```
struct Bits
{
   unsigned char  t0 : 1;
   unsigned char  t1 : 1;
   unsigned char  f1 : 1;
   unsigned char  f2 : 1;
   unsigned char     : 2;
   unsigned char  b1 : 2;
};
struct Bits cByte = {0,1,1,0,3};
```

Note gap (padding)

We do not include assignments for the ‘gap’

# What are they used for ?

Register settings, e.g.

- Many devices use a single register to set a series of values
- We could set/reset each bit but this would be very tedious
- Better to set a structure and the we can control each bit without affecting other bits

# Eg. - a typical engineering case (1)

Serial port control register

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| BR3 | BR2 | BR1 | DB2 | DB1 | SB2 | SB1 | P |

| | | |
|---|---|---|
| P: | Parity | (0=odd, 1 = even) |
| SB: | Stop bits | (0 bits,1 bit or 2 bits) |
| DB: | Data bits | (0=6 bits, 1=7 bits, 2 = 8 bit) |
| BR: | Baudrate | ( [x+1] * 1200 ), x= 0..7 |

# Eg. - a typical engineering case (2)

Serial port control register

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| BR3 | BR2 | BR1 | DB2 | DB1 | SB2 | SB1 | P |

P: Parity (0=odd, 1 = even)

DB: Data bits (0=6 bits, 1=7 bits, 2 = 8 bit)

SB: Stop bits (0=0 bits, 1=1 bit, 2=2 bits)

BR: Baudrate ( [x+1] * 1200 ), x= 0..7

To configure the port we would put zeros and ones in the relevant boxes and work out the decimal (or hex) value and assign this to the register e.g. for 9600,8,1,E

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 7=9600 | | | 2=8DB | | 1=1SB | | 1=E |

=243 (0xf3)

# Creating a structure to set register values

A bit field `struct` can help make this more manageable as we can separate items

```
struct RS232
{
   unsigned char  p : 1;  // parity bit
   unsigned char  sb : 2;  // stop bits
   unsigned char  db : 2;  // data bits
   unsigned char  baud : 3;  //baud rate
};
```

Assigning:

```
   struct RS232 serial = {1,1,2,7};
```

Or

```
   serial.p = 1;
   serial.sb = 1;
   serial.db = 2;
   serial.baud = 7;
```

# Improving even further…

Note: For REALLY good code we can use #define to create constants for the various parameters and use these in our code.

This makes it very easy to read and to update, consider our previous example…

```
P:  Parity      #define parity_odd        0
                #define parity even       1

DB:Data bits    #define data_bits_6       0
                #define data_bits_7       1
                #define data_bits_8       2

SB:Stop bits    #define stop_bits_0       0
                #define stop_bits_1       1
                #define stop_bits_2       2

BR:Baudrate     #define BAUD_1200         0
                #define BAUD_2400         1
                ….. 
                #define BAUD_9600         7
```

# Which is much easier to read

Giving

Assigning:

```
    struct RS232 serial = {parity_odd, stop_bits_1 , data_bits_2, BAUD_9600};
```

Or

```
    serial.p = parity_odd;
    serial.sb = stop_bits_1;
    serial.db = data_bits_2;
    serial.baud = BAUD_9600;
```

*Instead of*

Assigning:

```
    struct RS232 serial = {1,1,2,7};
```

Or

```
    serial.p = 1;
    serial.sb = 1;
```

*etc.*