Introduction to C

Session 1

Dr Mark Doughty, School of Engineering and Physical Sciences, University of Lincoln.

C features

- Some keywords – not many
- Pointers, arrays
- Compound data types:
  - Eg: Structures
- Compiles to native platform code
- Widely used
  - Around since 1972 (Dennis Richie, AT&T Bell Laboratories)
- Evolved over the years
  - C24 current standard

C uses

- Operating systems
  - Unix/BSD/Linux + flavours
- Web servers
  - Apache, etc
- Compilers/Interpreters
  - Python, Lua, GCC (mostly)
- Version control
  - Git, Subversion
- + embedded systems, games, etc

C and other languages

- C doesn’t have:
    - Exceptions/exception handling
    - Garbage collection
    - Object-oriented features

- However…
    - C is a low(er) level language
    - Produces fast(er) code

- Beware though…
    - No type safety/checking (you have to do this as the programmer)
    - No range checking (you have to do this as the programmer)
    - No memory management (you have to do this as the programmer)

Editing, running, debugging C

- Many configurations
  - I have used VS Code/MinGW gcc as the IDE/compiler
  - See https://code.visualstudio.com/docs/cpp/config-mingw
- gcc is also included with Linux
  - Check - open cmd prompt, enter `>gcc -version`
  - If not present, install it using the instructions above
- Make sure that gdb is installed to provide debugging

Editing, running, debugging C

- A basic use of gcc is:
  - `>gcc -Wall -o <output_exe> file1.c`

- So, for a c file called ‘virtualMachine.c’, and to produce an output of ‘vm.exe’:
  - `>gcc -Wall -o vm virtualMachine.c`

‘`-Wall`’ is the compiler option for ‘all warnings’.

So, the build will fail if there are no errors, but there are warnings

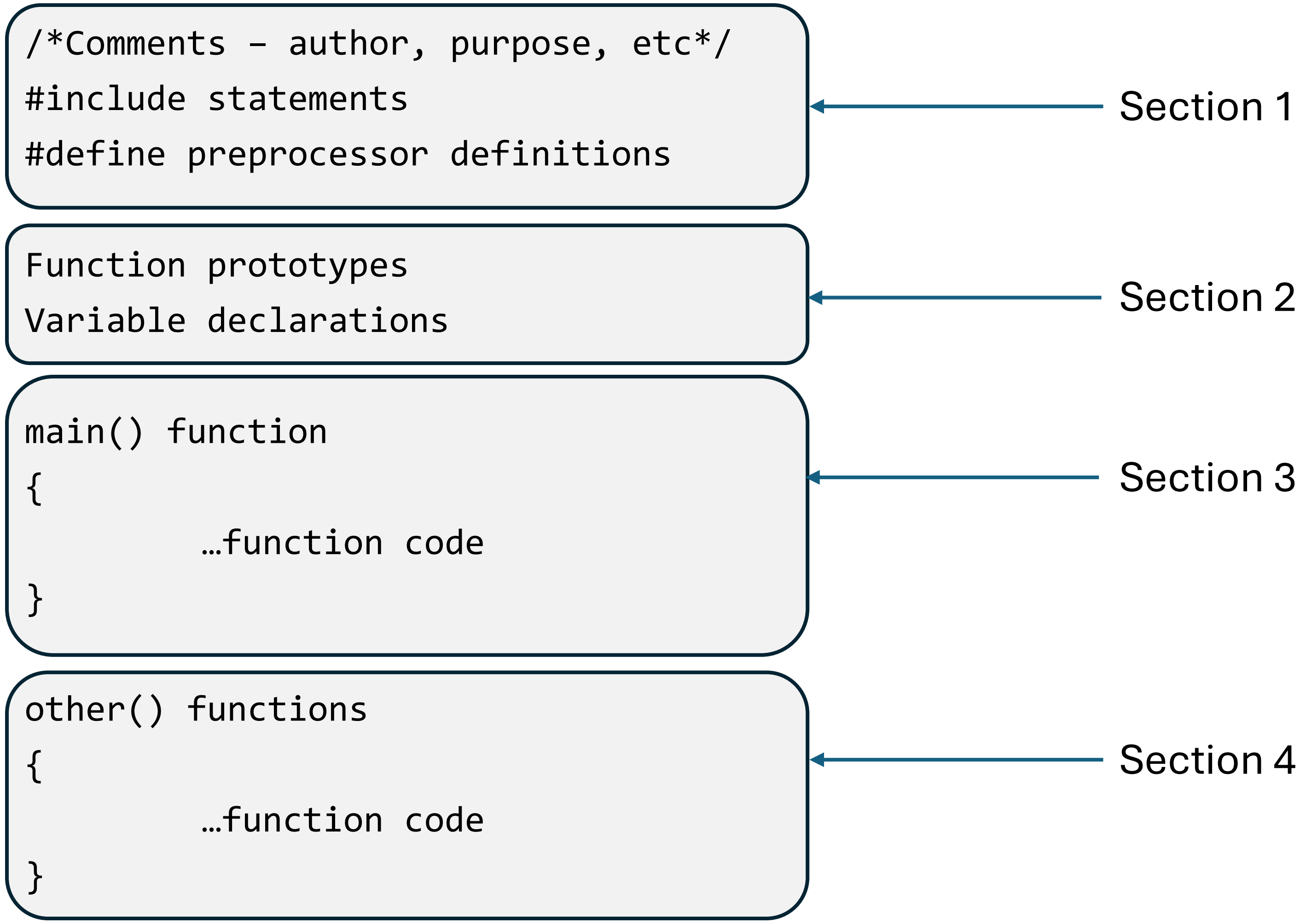
‘vm.exe’ can be run as normal from the command line

First C program

>Hello Lincoln!

Structure of a typical C file

>Will have a .c file extension



```
/*Comments – author, purpose, etc*/
#include statements
#define preprocessor definitions
```

```
Function prototypes
Variable declarations
```

```
main() function
{
        …function code
}
```

```
other() functions
{
        …function code
}
```

Section 1: Comments

- Can use:

```
//single line comment
/*single line comment*/
/*multiple
  line
  comment*/
```

Older compilers might not support these

- Can appear anywhere in the code

Section 1: `#include` statements

- `#include <stdio.h>`
  - Contain a set of predefined functions, variable declarations etc
  - Header file declared this way is in the ***system directory*** for header files
    - The ‘include’ directory for the compiler
- `#include “stdio_new.h”`
  - Again, contains a set of predefined functions, variable declarations
  - Header file declared this way is in the ***local source file directory***
    - ***The same directory as the .exe***

- The system directory for include files is assumed by default

Section 1: Preprocessor defines

- We can define constants to be used in the code:

```
#define var 10
```

Anywhere that ‘var’ is used in the code, it is defined/used as 10.
‘var’ cannot be modified.

- We can define functions here also:

```
#define x2(x,y) ((x+y)*2)
```

```
int q = 2;
int r = 5;
int answer = x2(q, r);
printf("%d", answer);
```

Section 2: Variable declarations

- In C, we must declare variables before they are used, eg:

```
int f;
int x = 3;
float y = 9.43;
char z = ‘q’;
```

Variables can also be initialised when they are created

- We can declare global-type variable in header files to use in multiple files

```
extern int globalVar;          /*Declaration in header file

int globalVar = 10;            /*Use in another file – but must define it here
```

Section 2: Function prototypes

- Functions must be declared before use*
- So, if we want to use a function called ‘output’ that takes no data, and returns nothing, then the prototype would be:

```
void output(void);
```

- Later in the code, we can now use the function as normal:

```
void output(void)
{
   /*something
}
```

*We don’t need a prototype, if the function is declared before it is used in `main()`

Section2: Function prototypes

- The general form of a function prototype is:

```
return_type function_name (arg1, arg2, arg3, ….);
```

Value returned from the function to the caller

Local variables which are passed to the function from the caller – which it needs to do its task

Either one can be `void`, meaning nothing is passed in and/or returned from the function

Section 3: The `main()` function

- There should only be one of these in the program.
- It is the start point for the executable.
- Can be:

```
int main(void){    ..    }
```

Returns 0 if it ran successfully, non-zero if it didn't and had an error.

```
int main(int argc, char **argv){   ..   }
```

When we pass in command line arguments, `argc` is the number of arguments + the program itself, and `argv` is the arguments (we'll do more on what the `char **argv` means later)

Section 3: The main() function

```
#include <stdio.h>
int main(void)
{
        printf(“Hello Lincoln”);

        puts(“Hello Lincoln”);

        putchar(‘w’);
}
```

Outputs the string to stdout (the console probably) without a new line, but if it contains ‘control characters’ (like ‘%’) you would get undefined behaviour – eg “Hello %Lincoln”

Outputs string to stdout with a new line at the end. It is a ‘string literal’, so any control characters are outputted with no effect.

Outputs a single character to stdout – no new line added. If string rather than character is specified – error.

Section 3: The `main()` function

- 'Special' characters can be added into the output statements to add formatting:
  - `\\` (a backslash)
  - `\b` (a backspace)
  - `\t` (a tab)
  - `\r` (a carriage return)
  - `\n` (a line feed)

'\' is called the 'escape character' and is essential if a special character or action is to be done.

'carriage return' is when the cursor (or 'caret') returns to the start of that line

'line feed' is when the cursor goes to the start of the next line.

Section 3: The `main()` function

- Try it!
- What would be the output from:

```
printf(“ab”);
printf(“\rcd”);
printf(“\be\nf”);
```

Section 3: The `main()` function

- The output would be:

```
ce
f
```

- Why?

“ab”	Prints ‘ab’ to the console
“\rcd”	Returns the cursor to the beginning of the line (carriage return), and then prints ‘cd’
“\be\nf”	Backspace (so deletes the ‘d’), outputs e, moves the cursor to the start of the next line (line feed), and outputs ‘f’

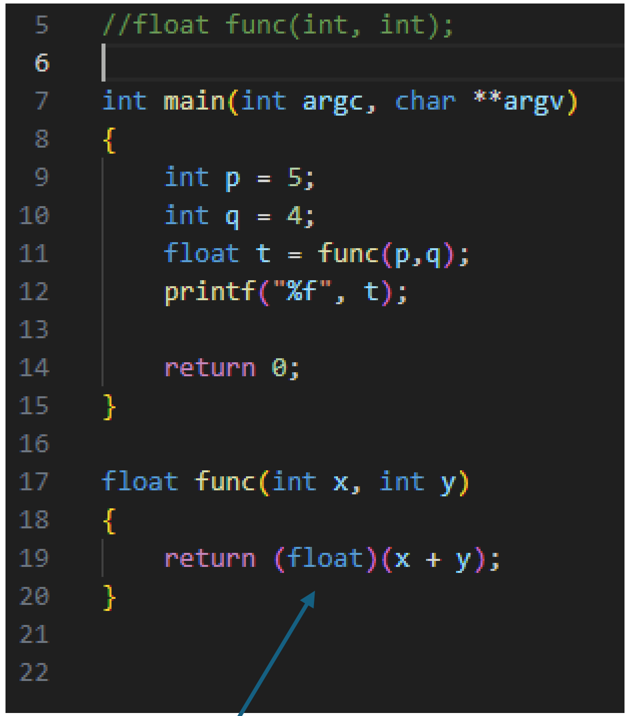
Section 4: Custom functions

- C allows us to create custom functions
  - A block of code, that only runs when it is called.
  - We need to 'prototype' the function first before we can use it* **
  - Declare its '**signature**' (return type, name, parameters)

- We can pass data into our function
  - '*Parameters*' which the function accepts are defined when the function defined
  - '*Arguments*' are the actual data values that are passed into the function

*Unless we declare the function first.
**Some modern compilers will 'assume' a function takes and returns an int – this is dangerous though.

Section 4: Custom functions



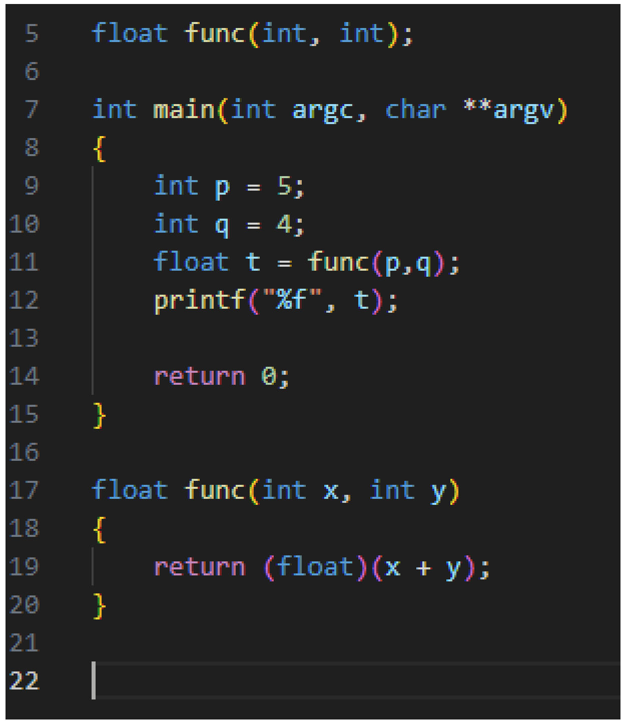
```c
//float func(int, int);

int main(int argc, char **argv)
{
    int p = 5;
    int q = 4;
    float t = func(p,q);
    printf("%f", t);

    return 0;
}

float func(int x, int y)
{
    return (float)(x + y);
}


```

Without prototype, the function is used before it is declared.

Doesn’t compile.

*This is a ‘cast’. It is a form of ‘explicit conversion, where you can force the datatype specified to be used.

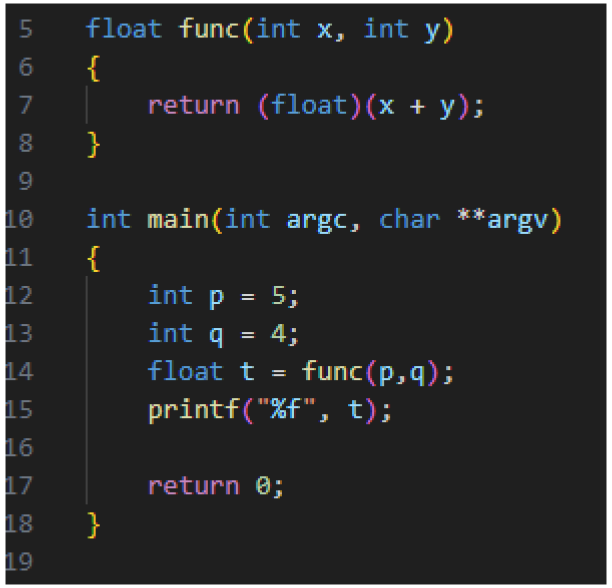
Section 4: Custom functions



```c
float func(int, int);

int main(int argc, char **argv)
{
    int p = 5;
    int q = 4;
    float t = func(p,q);
    printf("%f", t);

    return 0;
}

float func(int x, int y)
{
    return (float)(x + y);
}

```

Prototype is used this time, the function is used in main() using the functionality specified in the function.

Section 4: Custom functions



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

int main(int argc, char **argv)
{
    int p = 5;
    int q = 4;
    float t = func(p,q);
    printf("%f", t);

    return 0;
}
```

Function fully declared before it is used. Don’t need a prototype in this case.

In conclusion …

- In this session, we have covered:
  - C features and uses
  - Comments
  - Preprocessor statements
  - #include
  - Variable declarations
  - Output functions (`printf(), puts(), putchar()`)
  - Control characters/escape character
  - Function prototyping