University of Nottingham
UK | CHINA | MALAYSIA

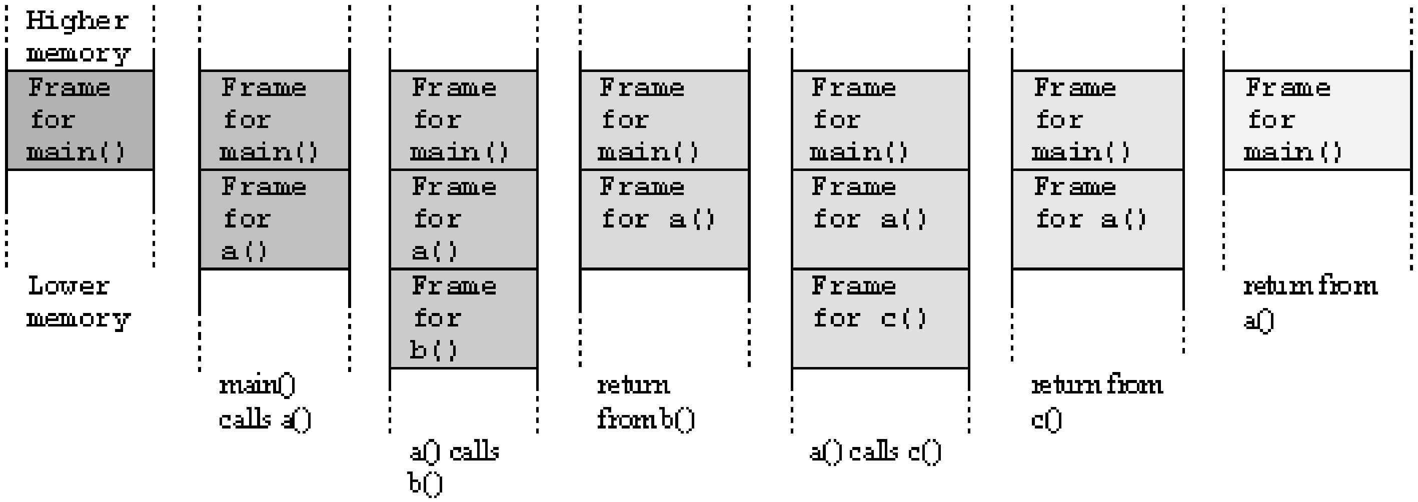
Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

Recap on functions and memory

http://www.tenouk.com/ModuleZ.html

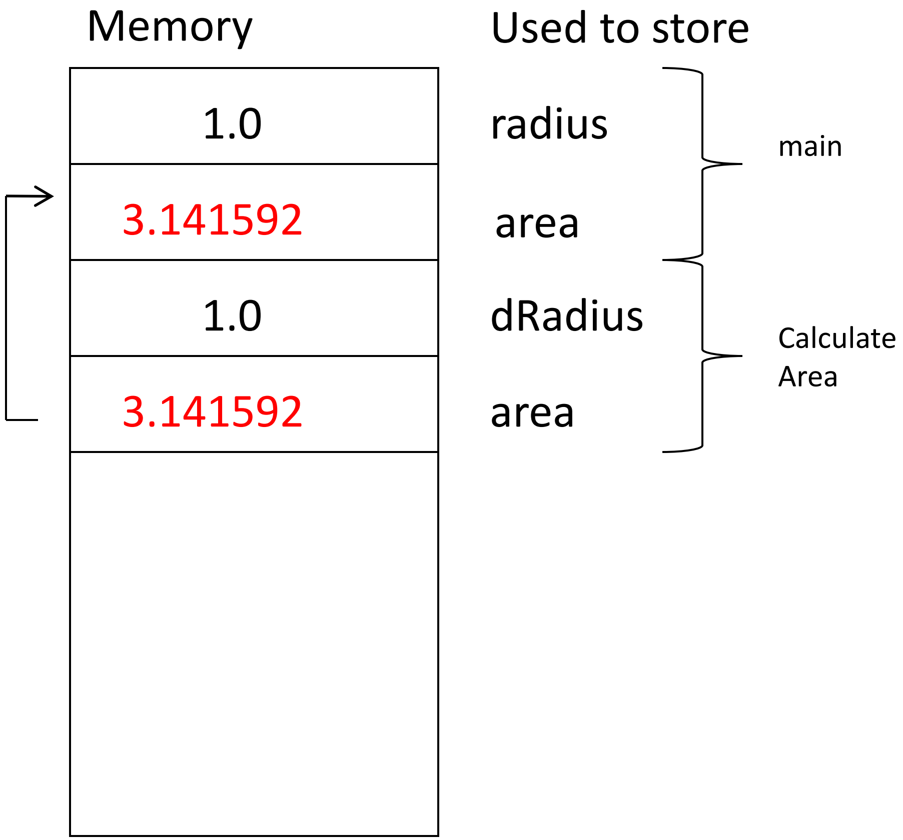
```
int a (void );  // Function prototypes
int b (void );  // memory is not allocated until
int c (void );  // functions are actually used

int a( void)
{
   b();
   c();
   return 0;
}

int b( void )
{
   return 0;
}

int c( void )
{
   return 0;
 }

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



A ‘Frame’ is the term for the block of memory used by a function

Copies of variables are created when function called

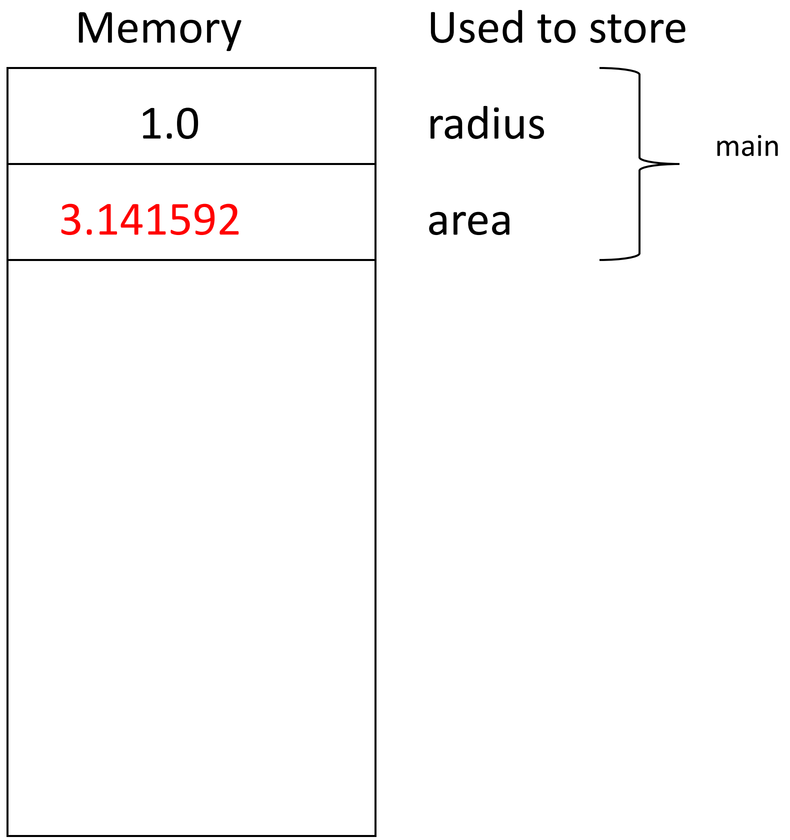
```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.141592 * dRadius * dRadius;
    return ( area);
}
```



Memory	Used to store	
1.0	radius	main
3.141592	area	
1.0	dRadius	Calculate Area
3.141592	area	

Memory is released on return from function

```
double CalculateArea ( double );

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    area = CalculateArea (radius);
    return 0;

}


// And here is our function

double CalculateArea ( double dRadius )
{
    double area;
    area = 3.14159265 * dRadius * dRadius;
    return ( area);
}
```



Memory	Used to store	
1.0	radius	main
3.141592	area	

Lab 1 Programming Assignment

Submit a single .zip file named Lab1PrepXXX.zip, where XXX are your initials

Submission deadline: **3pm, Thursday 26th October**

Anything submitted after 3.01pm will get a late submission penalty – upload in plenty of time to avoid technical glitches!

Introduction

- Today we will cover:
- Chapter 13 – Pointers – part 1
- Chapter 14 – Functions – part 2
- Chapter 15 – Pointers part 2 – using with arrays
- Chapter 20 – Preprocessor directives

Start recording!!

Chapter 13

Pointers: Part 1

Pointers (1)

This is a key to C but something that needs careful thought

Do not worry if you do not grasp it first time

It is something that needs ‘contemplation’

Pointers (2)

A pointer is a special type of variable in which we store a **memory address**

Pointers have the ‘ability’ to know how much storage (in bytes) the item to which they are pointing takes up in memory

Pointers (3)

Pointers are used (in particular)

- For Dynamic Arrays
- For Speed (both in Arrays and when calling functions)
- To Maximize the use of memory

When using pointers we go directly to a memory address

- Note: When we get/set ***a*** variable, e.g. a=10 the system does this for us, looking up the memory address of ***a*** and then storing the value 10 at that location.

Pointers: the basics

For each variable type in C we can declare a pointer

Initially a pointer points ‘nowhere’

It is then up to us (as programmers) to create the code that assign to the pointer to the address of

- Variable,
- Array or
- Function (we will not be doing this!)

Pointers: in practice

There are three ‘steps’ [*] in using a pointer when accessing an existing variable

- Create the pointer
- Assign to it the address of an existing variable
- Use the pointer

- [*] Note: There is a 4th step when using pointers with arrays, we cover this later

Pointers: Create

We define a pointer in much the same way as any variable, the only difference is we precede the variable name with a *

Here are a few examples

```
int *i;
char *c;
float *f;
```

Pointers: Assign

If we have an existing variable we can request its address with the & operator (you have been using this in *scanf*)

Since this an address of the variable, we can then assign it to a pointer that has been created to store such an address

```
int *i;
i = &Data;

char *c;
c = &Letter;

float *f;
f = &fVar;
```

Memory Address	Memory	Used to store
2340	10	Data
2344	‘y’	Letter
2346	22.3	fVar
	2340	i
	2344	c
	2346	f

Remember: Pointer type must match variable type

Pointers: Accessing

Once we have our pointer ‘pointing’ to a memory address we can ‘access’ that address to get/set values.

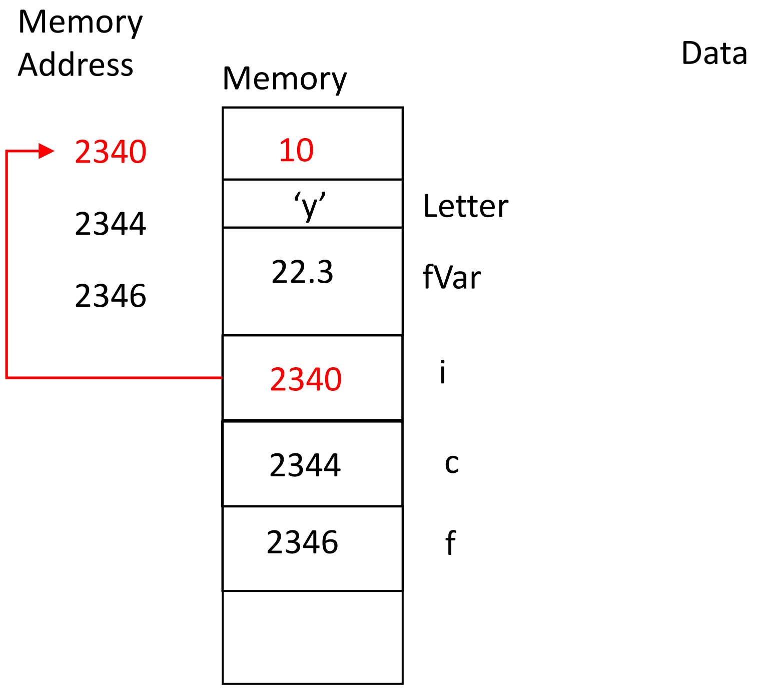
To indicate that we want to access the memory address stored in a pointer (the formal term for which is pointer dereferencing) we again use the *

A few examples:

```
printf("%d", *i );  // Get the item at the memory address stored in i
printf("%c", *c);   // Get the item at the memory address stored in i
int j = *i;         // Get the item at the memory address stored in i
                    // and store in j
*i = 72;            // Store the value 72 in the memory address stored in i
```

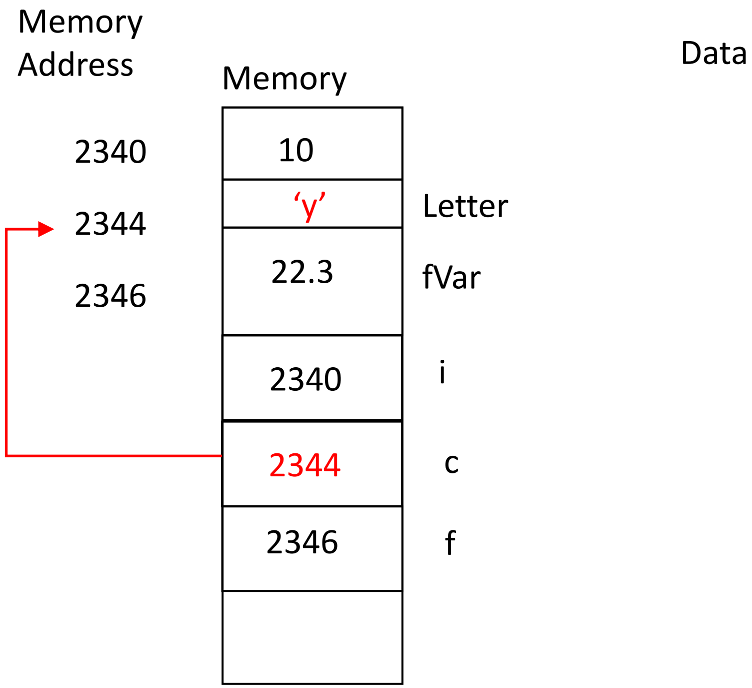
Accessing pointers illustrated (1)

```
printf("%d", *i );      // Get the item at the memory address stored in i
printf("%c", *c);       // Get the item at the memory address stored in i
int j = *i;             // Get the item at the memory address stored in i and store in j
*i = 72;                // Store the value 72 in the memory address stored in i
```



Accessing pointers illustrated (2)

```
printf("%d", *i );     // Get the item at the memory address stored in i
printf("%c", *c);      // Get the item at the memory address stored in i
int j = *i;            // Get the item at the memory address stored in i and store in j
*i = 72;               // Store the value 72 in the memory address stored in i
```



Memory Address

Memory

Data

2340 10

2344 'y' Letter

2346 22.3 fVar

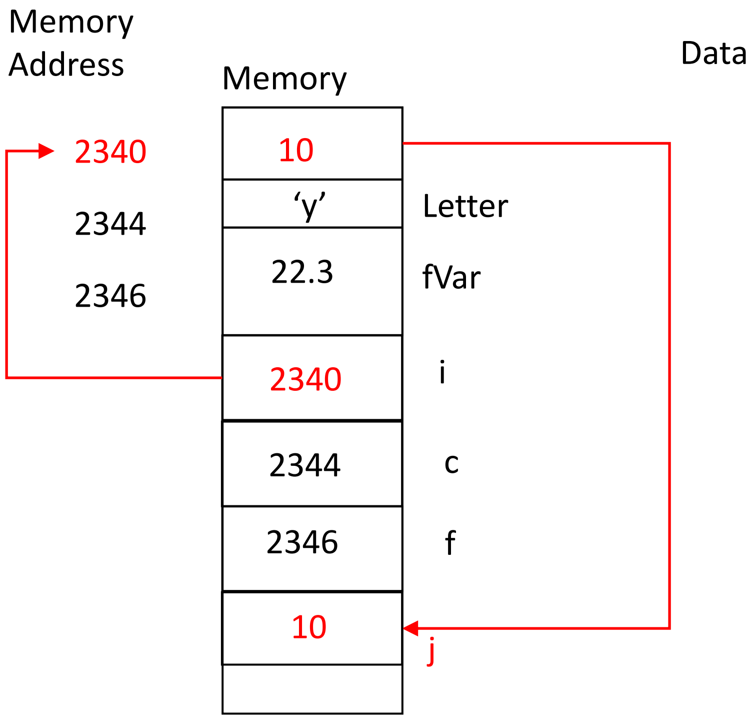
2340 i

2344 c

2346 f

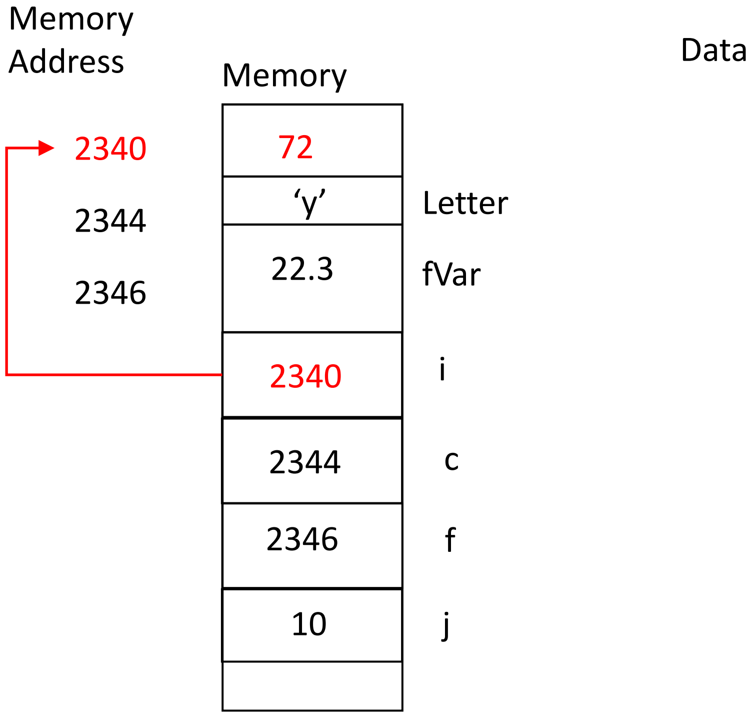
Accessing pointers illustrated (3)

```
printf("%d", *i );    // Get the item at the memory address stored in i
printf("%c", *c);     // Get the item at the memory address stored in i
int j = *i;           // Get the item at the memory address stored in i and store in j
*i = 72;              // Store the value 72 in the memory address stored in i
```



Accessing pointers illustrated (4)

```
printf("%d", *i );     // Get the item at the memory address stored in i
printf("%c", *c);      // Get the item at the memory address stored in i
Int j = *i;            // Get the item at the memory address stored in i and store in j
*i = 72;               // Store the value 72 in the memory address stored in i
```



Pointers: In summary

1: Declare variable, e.g.

```
int a,b;
float c,d;
```

2: Declare pointer variables

```
int *p, *q;
float *r, *s
```

3: Assign memory addresses of variables to pointers

```
p = &a ; q = &b ; r = &c; s = &d;
```

4: Pass them or use them !

```
scanf("%d", p)
*q = 7;
```

← Note, same as: `scanf("%d",&a);`

C13/accessing_via_pointers.c

Pointers: A quick test

What is the value in C following the last executed line?

```
int  a=1, *b, c ;     /* Define variable types,
                           initialise a to 1 */

b = &a;               /* b contains the address of a */

c = *b + 1;           /* c contains what ? */
```

Chapter 14

Function Programming (Part 2)

Functions - a review

Functions - a quick reminder

Live outside of ‘main()’ code

- They can be in other source files if you wish
- Have (if required) a prototype in a header file

Functions

All Functions consist of three parts:

- Return type

  Any valid C variable type or void
- Name

  A name of your choice
- Argument list

  At least one valid C variable (or void); multiple ones are separated by commas

Plus some code of course!

We remember too, a function can return, <u>at most,</u> one thing

Now consider this..

Often when we do a calculation we may want to know a number of things:

- For a set of numbers we might wish to know the minimum, maximum & average
- For a quadratic equation, the two possible solutions

We know we can write functions to do these tasks, but a function can only return one value

So how do we get round this?

Well...

We could write functions to do each of the tasks individually

- But this would be very inefficient
- e.g.
  - To find the minimum, max & average of an array of number we would have to loop across all the variables three times
  - Better (more efficient) to loop once and do all three tasks at the same time

There is a solution ☺

- And solution is to use pointers!

Functions using pointers for parameters

By Reference

- Rather than pass variables to functions, we pass the memory address where variable(s) are stored
- Remember: To obtain the memory address of variable we prefix it with &

Exactly as we have been doing with *`scanf`*

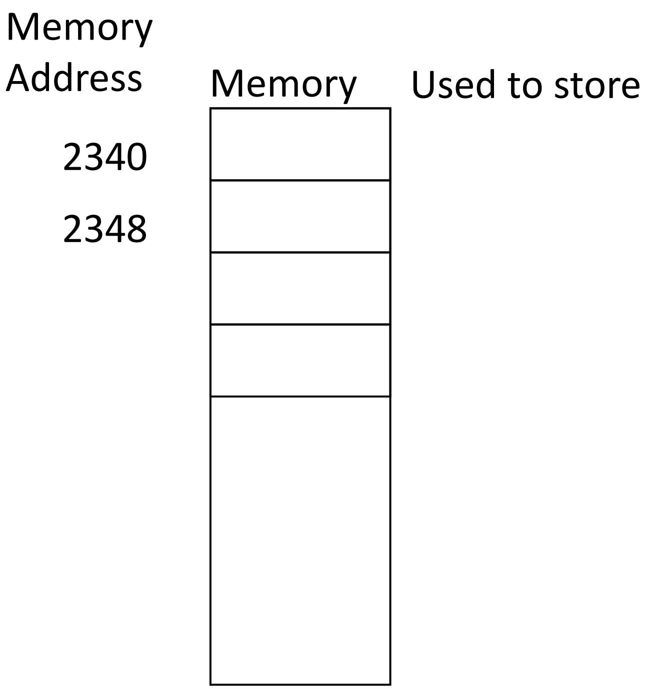
This ‘gets round’ the problem of

- Variables being ‘private’ to functions

So back to the pictures...

By reference: note the & and * (1)

```
void CalculateArea ( double Radius, double *pArea);  // note the ‘*’

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    CalculateArea (radius, &area);
    return 0;

}


// And here is our function

void CalculateArea ( double Radius, double *pArea )
{
    *pArea = 3.14159265 * Radius * Radius;
    return;
}
```



Memory Address

Memory

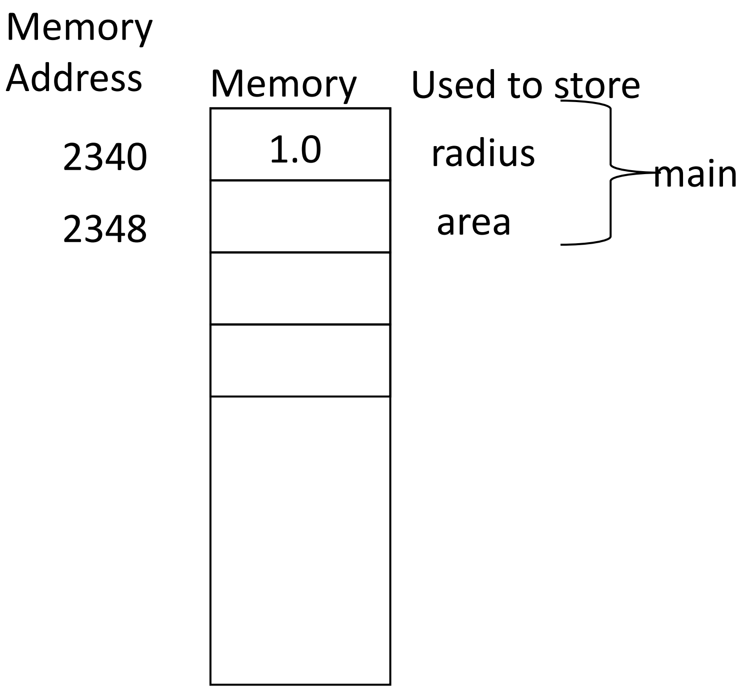
Used to store

2340

2348

By reference: note the & and * (2)

```
void CalculateArea ( double Radius, double *pArea);  // note the ‘*’

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    CalculateArea (radius, &area);
    return 0;

}


// And here is our function

void CalculateArea ( double Radius, double *pArea )
{
    *pArea = 3.14159265 * Radius * Radius;
    return;
}
```



Memory Address	Memory	Used to store
2340	1.0	radius
2348		area

By reference: note the & and * (3)

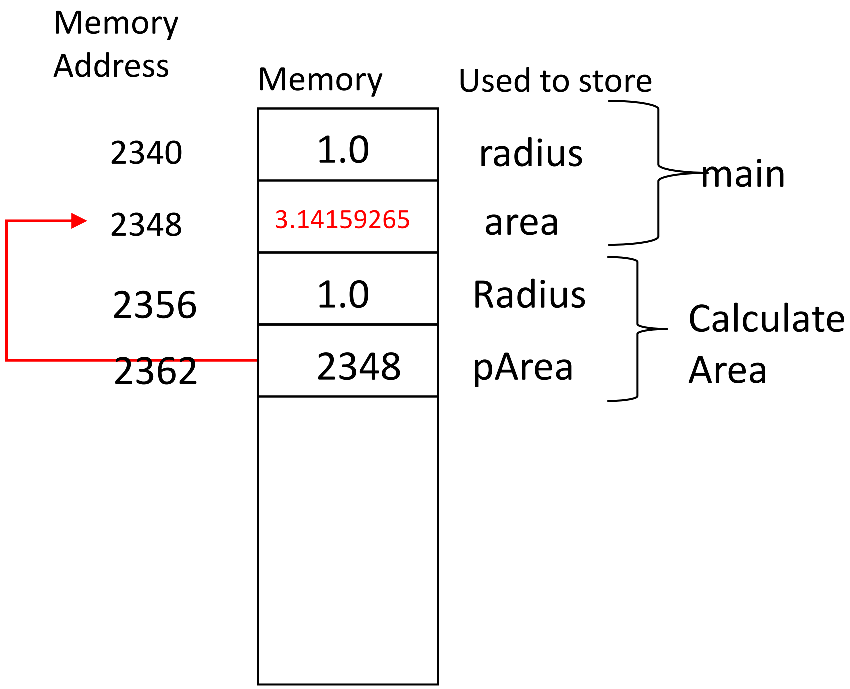
```
void CalculateArea ( double Radius, double *pArea);  // note the ‘*’

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    CalculateArea (radius, &area);
    return 0;

}


// And here is our function

void CalculateArea ( double Radius, double *pArea )
{
    *pArea = 3.14159265 * Radius * Radius;
    return;
}
```

Values/Addresses of the variables (which are COPIED to the function)

Memory Address	Memory	Used to store	
2340	1.0	radius	main
2348		area	
2356	1.0	Radius	Calculate Area
2362	2348	pArea	

By reference: note the & and * (4)

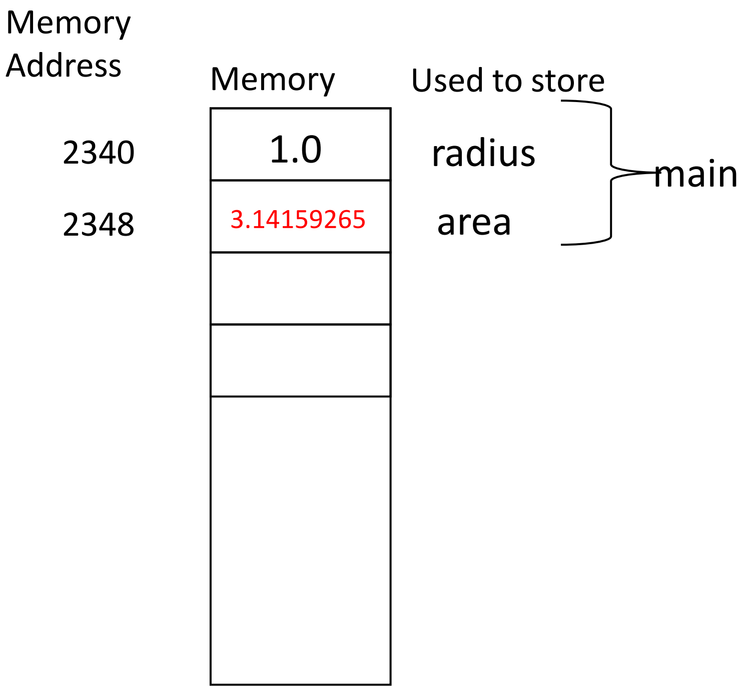
```
void CalculateArea ( double Radius, double *pArea);  // note the '*'

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    CalculateArea (radius, &area);
    return 0;

}


// And here is our function

void CalculateArea ( double Radius, double *pArea )
{
    *pArea = 3.14159265 * Radius * Radius;
    return;
}
```



The * here mean we access the memory locations (to get/set values)
The ‘others’ are multiplication signs

By reference: note the & and * (5)

```
void CalculateArea ( double Radius, double *pArea);  // note the ‘*’

// This is the main code for our application

int main()
{
    double radius, area;
    radius = 1.0;
    CalculateArea (radius, &area);
    return 0;

}


// And here is our function

void CalculateArea ( double Radius, double *pArea )
{
    *pArea = 3.14159265 * Radius * Radius;
    return;
}
```



Memory Address	Memory	Used to store	
2340	1.0	radius	main
2348	3.14159265	area	

LC14\pointer_function_example_1.c

Memory addresses of Variables (1)

Please note:

- In the previous few slides the memory addresses were ‘made up’ for purposes of the example
- We cannot predict the memory address that a variable will be stored at
- The ‘&’ solves the problem by providing us with the memory address at which a variable is being stored

Memory addresses of Variables (2)

Expanding the process:

- In the 1st example demonstrated we were only calculating one thing
  - As such this was a rather ‘odd’ way to get a result (we could have just used the return value)
- The reason for this approach becomes more obvious when we aim to calculate multiple things in the same function
  - e.g. the volume and surface area of a cylinder

LC14\pointer_function_example_2.c

Summary so far…

When we call a function we (if needed) can pass two types of parameters

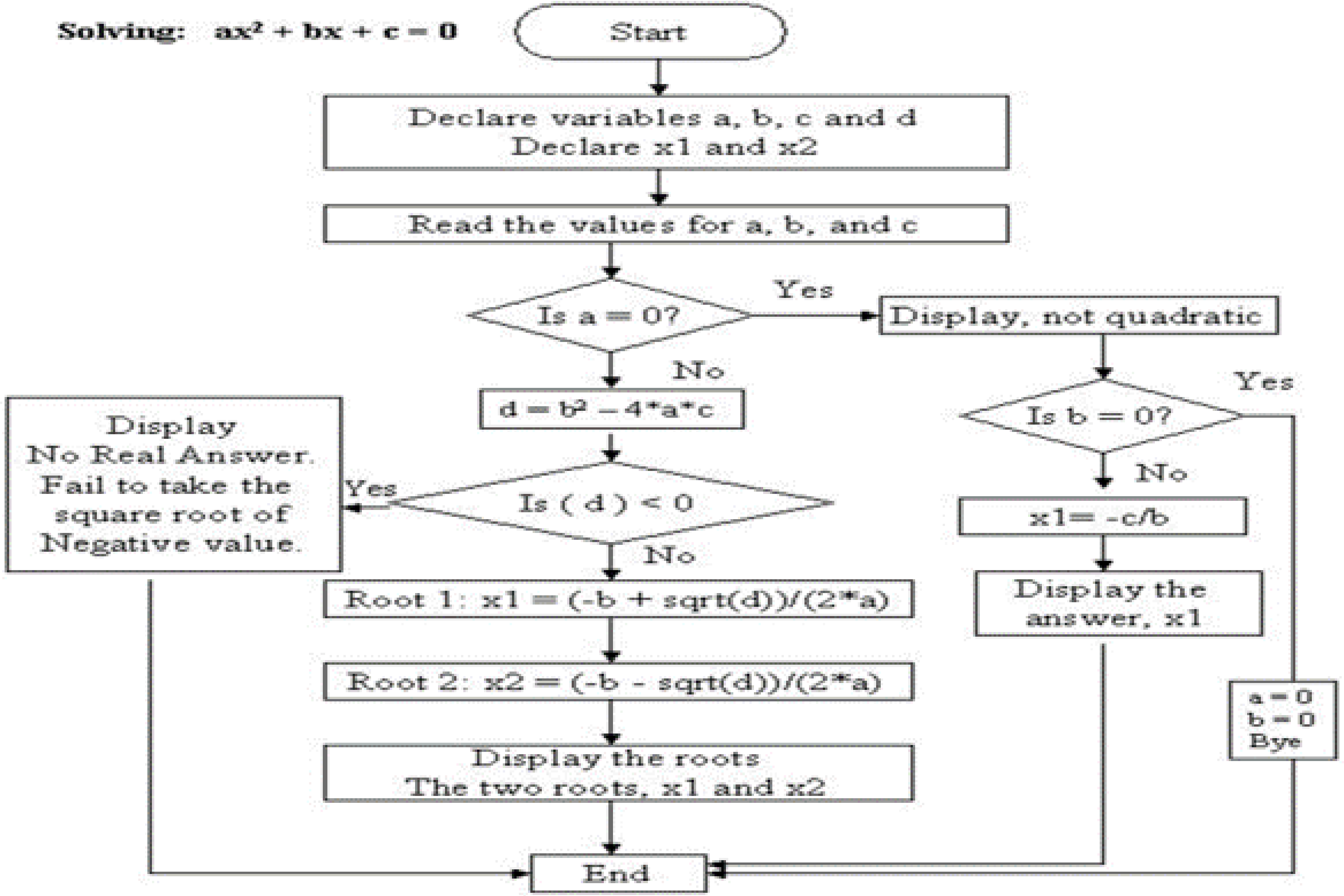
- Formal: Values (either variables or ‘actual’ values – eg 3.0, ‘a’ etc.).
- References: The location in MEMORY where VARIABLES are being stored

In either case please remember

- The parameters passed (formal or reference) are **COPIED** into new variables that exist in memory as long as the called function runs

Combining Pointers & Functions (1)

Now we have considered this, let us consider how we apply this to our quadratic equation solver



Combining Pointers & Functions (2)

Now we have considered this, let us consider how we apply this to our quadratic equation solver

- We know there are input parameters: a,b and c
- The outputs are: x1 and x2

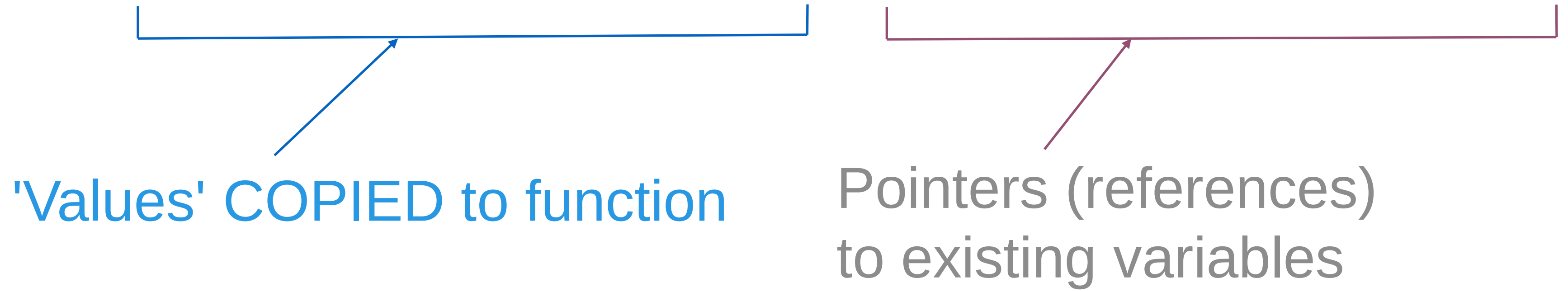
So how do we define this function?

- For the inputs we can use formal parameters (pass actual values)
- For the outputs, pass references to existing variables into which the calculated values can be placed

Combining Pointers & Functions (3)

Which gives us

```
void SolveQuadratic ( float a, float b, float c, float *x1, float *x2 )
```



'Values' COPIED to function

Pointers (references)
to existing variables

Remember: All things are copied to functions – even memory addresses

Combining Pointers & Functions (4)

We can improve this a little….

By having a return value that indicates the success (or otherwise) of the function.

We might return:

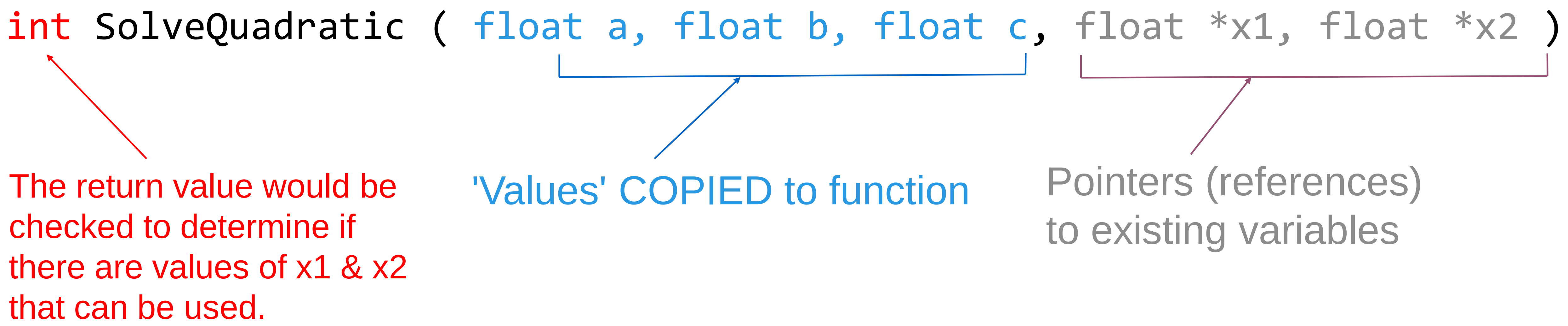
- 0: All is OK, the values in x1 and x2 are the solutions for the supplied values of a,b, and c
- -1 The values supplied were not valid for a quadratic (e.g. a=0)
- -2 The solution is complex and cannot be solved using this function

When using this function, we would examine the return value and ONLY use the values of x1 & x2 if a value of zero was returned.

This would modify the function to be:

Combining Pointers & Functions (5)

Which gives us



```
int SolveQuadratic ( float a, float b, float c, float *x1, float *x2 )
```

The return value would be checked to determine if there are values of x1 & x2 that can be used.

'Values' COPIED to function

Pointers (references) to existing variables

Remember: All things are copied to functions – even memory addresses

Chapter 15

Pointers: Part 2

Pointers and Arrays (1)

Before we start, a quick reminder…

When we create an array, the memory allocated is a continuous block (it has to be so that the nth item can be found)

Each item in the array has its own address which we can obtain, e.g. `&MyArray[n]`

Or we can calculate it as, address of nth item:

Address of nth item = `&MyArray [0] + (n * sizeof (array_type) )`

Address of nth item = `MyArray + (n * sizeof (array_type) )`

Reminder:

The name of an array is also the address of the 1st item (index [0])

Pointers and Arrays (2)

Since each item in an array has its own address, we could

- Create an array of pointers of the same size as the array
- Assign each pointer to its corresponding array item
    - e.g. PointerArray[n]= &Array[n]
- Use this array of pointers to access items

This would be a rather ‘pointless’ task as we could just as easily use the original array

There is however no need to do this, just one pointer is enough ☺

Pointers and Arrays (3)

It works as pointers can be indexed like an array – we just drop the asterisk

e.g. If we defined an array as

```
    int MyArray[20];
```

And created a pointer which we ‘point’ to the 1st address (`index [0]`)

```
    int *pArray = &MyArray[0];        (or int *pArray = MyArray; )
```

To get the nth item from the array we can use EITHER

```
    MyArray[n] OR   pArray[n]
```

Pointers and Arrays: Getting the 1st item

We know from using pointers with single variables that we can obtain the value at a memory address using the asterisk (pointer dereferencing)

If we again consider

```
int *pArray = &MyArray[0];          (or  int *pArray = MyArray; )
```

To get the zero item from the array we can use EITHER

```
MyArray[0]    OR     pArray[0]
```

But we could also use: `*pArray`

LC15\pointer_to_array_1.c

Pointers and Arrays: Getting the nth item (1)

Why use this approach?

We can do a ‘clever’ trick to move to the **next** item in the array

```
*pArray++;
```

This is VERY quick as it adds the *sizeof* the item to which it points to the current memory address (a single addition), e.g.

```
new_address = current_address + sizeof(array type)
```

rather than having to do the more complex calculation

```
new_address = base_address + ( n * sizeof(array type) )
```

Note: We can also move backward through an array, e.g. *pArray–-;

Pointers and Arrays: Getting the nth item (2)

NOTE:

When using this approach you still need to be careful that

your code does not go beyond the bounds of the array (forwards or backwards)

LC15\pointer_to_array_2.c

Pointers and Arrays: Getting the nth item (3)

It is also possible to use an index when using this approach, e.g.

```
*(pArray+n);
```

However this can get confusing – you may as well use:

```
pArray[n];
```

Pointers and Arrays: Getting the nth item (4)

The table below shows the different ways we can access array elements

Array index	From array	Pointer approach 1	Pointer Approach 2	Pointer Approach 2
0	MyArray[0]	pArray[0]	*pArray	*pArray
1	MyArray[1]	pArray[1]	*(pArray+1)	*pArray++
2	MyArray[2]	pArray[2]	*(pArray+2)	*pArray++
2	MyArray[3]	pArray[3]	*(pArray+3)	*pArray++
n	MyArray[n]	pArray[n]	*(pArray+n)	🟥

This would only follow if we were moving through the array
We cannot move to an arbitrary location using this approach

Chapter 20

Preprocessor Directives

Preprocessor Directives

There are three types we consider

- #include
- #define
- Code formatting (`#ifdef, #if` etc.)

Preprocessor Directives - #include

`#include`

- Inserts the contents of another file
  - This can be a header file associated with a standard library or one we have created ourselves
  - `#include <stdio.h>` - The <> brackets tell the compiler to search the path for the file, typically one of the standard libraries
  - `#include “funcs.h”` – The “” indicate that the file will be found in the current folder

Preprocessor Directives – #define

#define (We will cover this in more detail in Chapter 19)

- Allows us to define a label which we can use in code

- This is then substituted before compiling, e.g.

  ```
  #define M_PI 3.14159265358979323846
  ```

- In code we can then write (say)

  ```
  Area = M_PI * Radius * Radius;
  ```

- What is compiled is

  ```
  Area = 3.14159265358979323846 * Radius * Radius;
  ```

Preprocessor Directives : Formatting (1)

On occasions we may require code that:

- Has a debug version that output additional information but which we never wish to release
- Has a ‘demo’ version with reduced features

For both of these however we wish only to maintain one set of code file(s)

- We do not however wish to have to add/remove comment blocks each time to change modes

We can achieve this using pre-processor directives to select which code is compiled

- So just having one version!

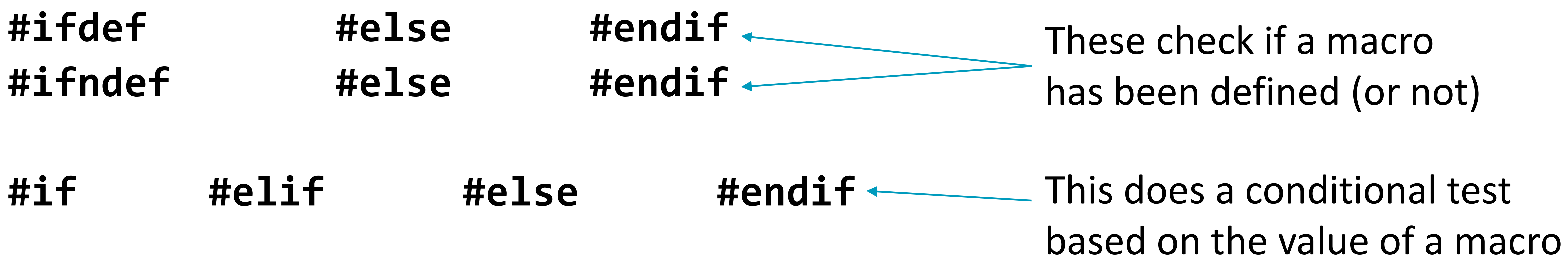
Preprocessor Directives : Formatting (2)

We do this using conditional statements at the pre-processor stage

The format is much like if/else if/else

The difference is we prefix with a # (and slightly change the commands)

The sets we can use are



```
#ifdef          #else      #endif
#ifndef         #else      #endif

#if      #elif      #else      #endif
```

These check if a macro has been defined (or not)

This does a conditional test based on the value of a macro

Preprocessor Directives : Formatting (3)

Consider:

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

#define DEBUG_ON 1

int main(void)
{
#ifdef DEBUG_ON
    printf("Debug mode - about to do something\n");
#else
    print("Running in standard mode");
#endif

    return 0;
}
```

As we have defined DEBUG_ON

Preprocessor Directives : Formatting (4)

Consider:

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

#define DEBUG_ON 1

int main(void)
{
#ifdef DEBUG_ON
    printf("Debug mode - about to do something\n");
#else
    print("Running in standard mode");
#endif

    return 0;
}
```

As we have defined DEBUG_ON

This condition is true

This line is included in the code to be compiled

Preprocessor Directives : Formatting (5)

Consider:

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

#define DEBUG_ON 1

int main(void)
{
#ifdef DEBUG_ON
    printf("Debug mode - about to do something\n");
#else
    print("Running in standard mode");
#endif

    return 0;
}
```

As we have defined DEBUG_ON

This condition is true

This line is included in the code to be compiled

This line is excluded (and will appear greyed in VSCode)

Preprocessor Directives : Formatting (6)

So what is actually compiled is…

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

#define DEBUG_ON 1

int main(void)
{
   printf("Debug mode - about to do something\n");
   return 0;
}
```

C20\formatting_directive_example.c

Preprocessor Directives : Formatting (7)

We can also use the
 #ifndef

This will test if a macro is NOT defined and compile if this is the case

As such it works in the opposite to the #ifdef directive

Preprocessor Directives : Formatting (8)

The #if version checks the value of a declared macro, e.g.

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

#define DEBUG_ON 1

int main(void)
{
#if DEBUG_ON == 1
   printf("Debug mode %d about to do something\n“, DEBUG_ON);
#else
   print("Running in standard mode");
#endif

   return 0;
}
```

It is still possible to use the macro value (as shown above)

C20\conditional_directive_example.c, RobotWriter5.0_Skeleton – serial.c