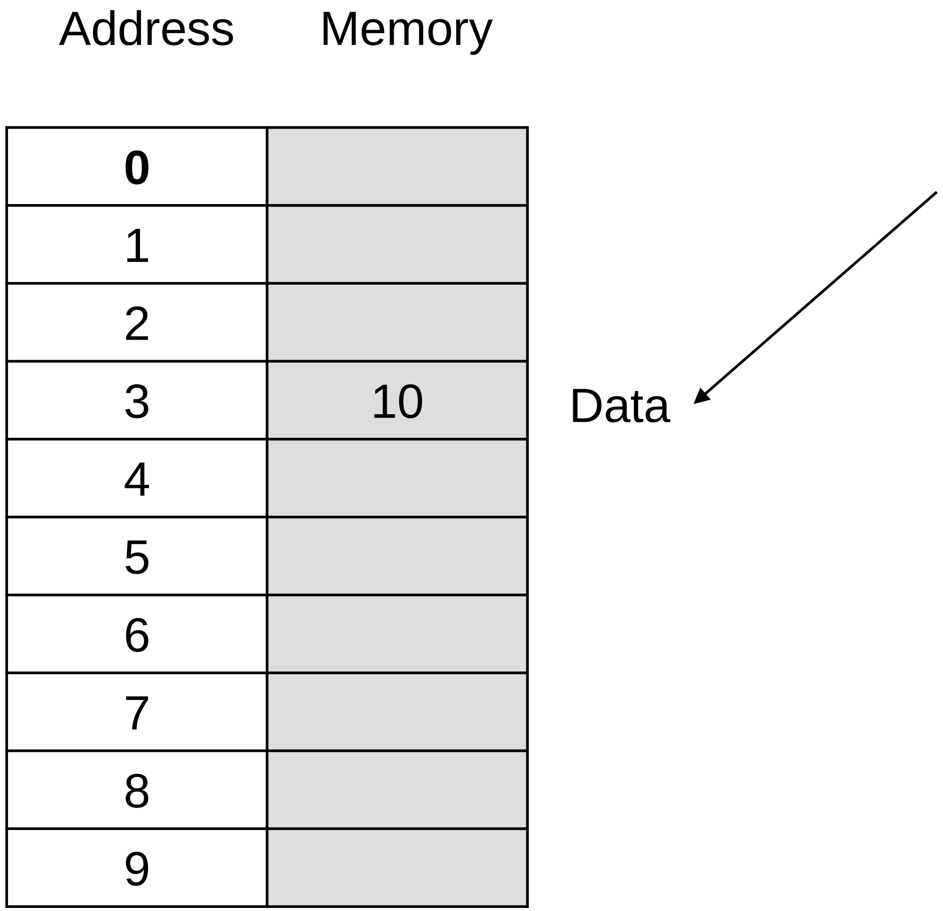
University of Nottingham
UK | CHINA | MALAYSIA

Computer Engineering and Mechatronics MMME3085

Dr Louise Brown

Pointers recap (1)

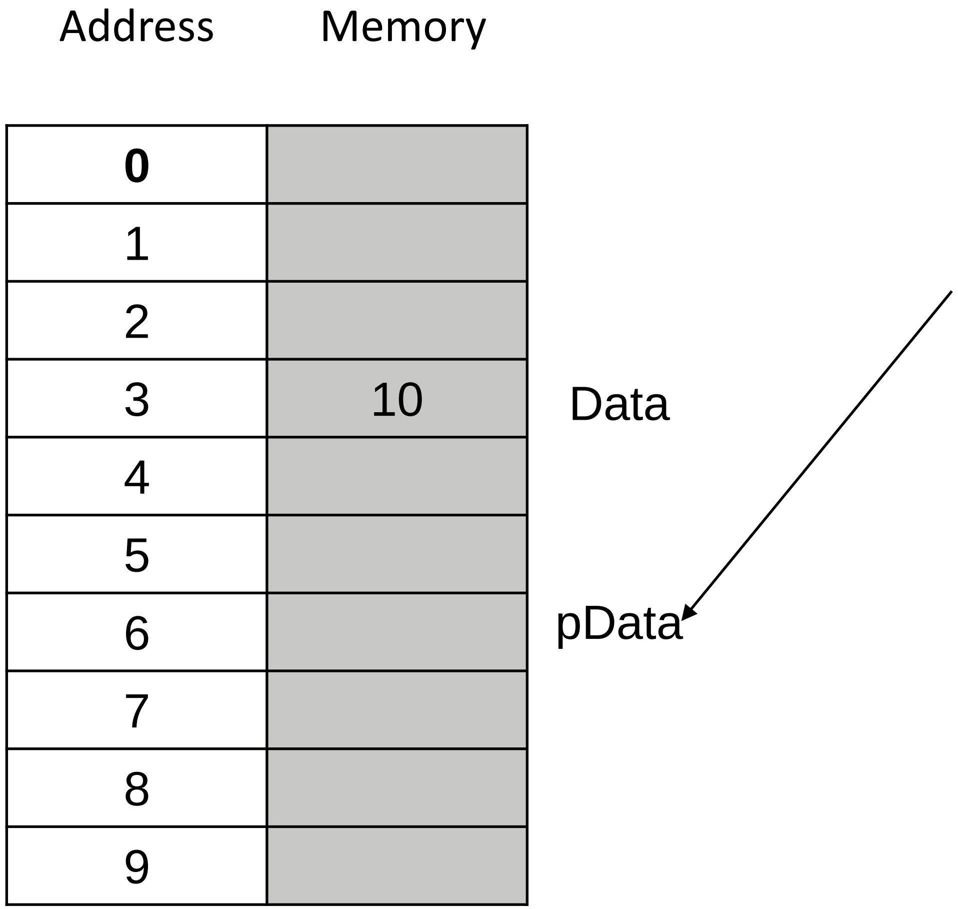


Address	Memory
0	
1	
2	
3	10
4	
5	
6	
7	
8	
9	

```
char Data = 10;
```

We have no control over where the system stores this

Pointers recap (2)



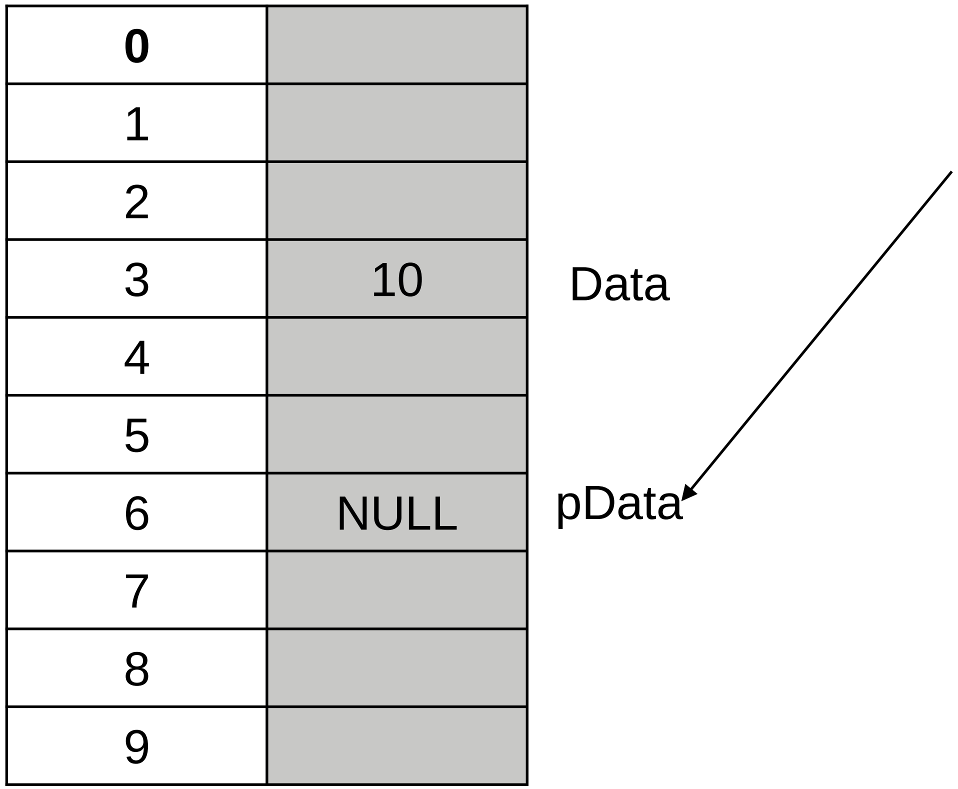
Address	Memory	
0		
1		
2		
3	10	Data
4		
5		
6		pData
7		
8		
9		

```
char Data = 10;
char *pData;
```

No value assigned to this pointer at the moment

It could contain any random data

Pointers recap (3)

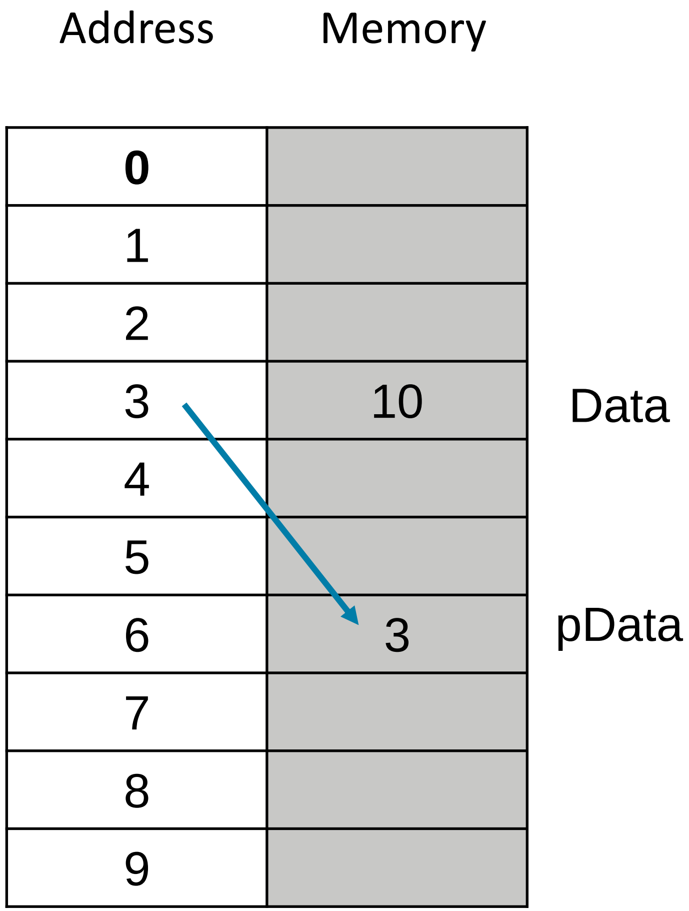


Address	Memory	
0		
1		
2		
3	10	Data
4		
5		
6	NULL	pData
7		
8		
9		

```
char Data = 10;
char *pData = NULL;
```

This is much safer!

Pointers recap (4)

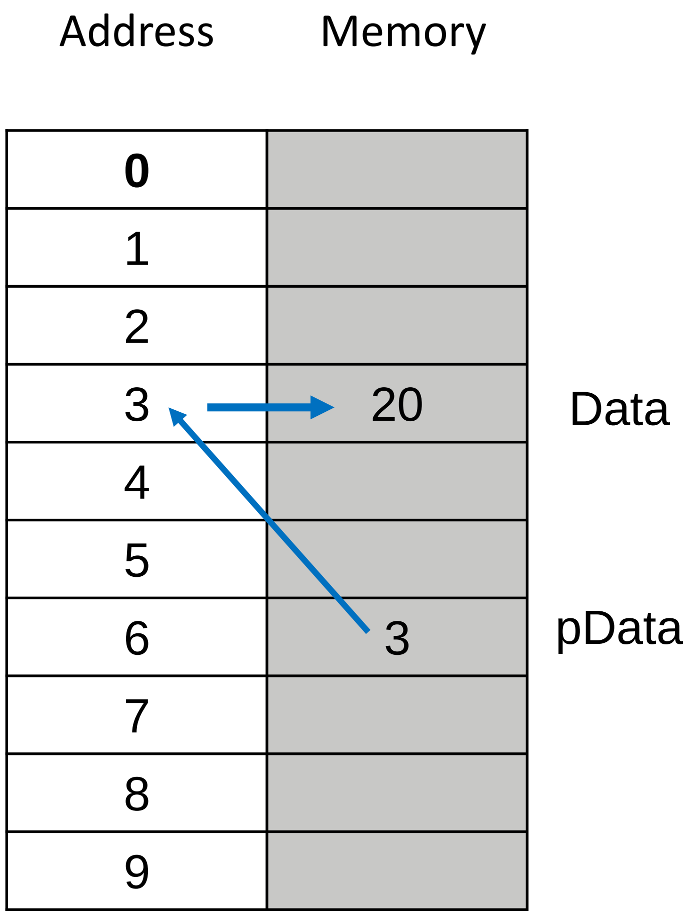


Address	Memory	
0		
1		
2		
3	10	Data
4		
5		
6	3	pData
7		
8		
9		

```
char Data = 10;
char *pData = NULL;
pData = &Data;
```

Use & to get the address of a variable

Pointers recap (5)



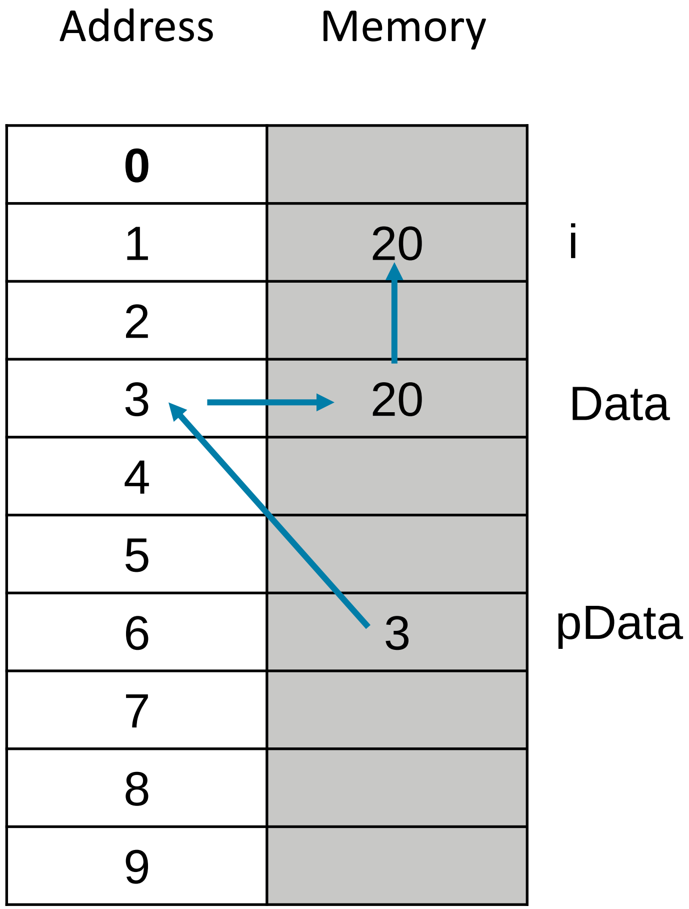
Address	Memory	
0		
1		
2		
3	20	Data
4		
5		
6	3	pData
7		
8		
9		

```
char Data = 10;
char *pData = NULL;
pData = &Data;
*pData = 20;
```

Pointer dereferencing using *

or 'what pData is pointing at' = 20

Pointers recap (6)



Address	Memory	
0		
1	20	i
2		
3	20	Data
4		
5		
6	3	pData
7		
8		
9		

```
char Data = 10;
char *pData = NULL;
pData = &Data;
*pData = 20;
```

```
char i = *pData;
```

i = ‘What pData is pointing at’

Introduction

- Today we will cover:
- Chapter 16 – Dynamic memory allocation
- Chapter 17 – Function programming (part 3)
- Chapter 19 – Advanced data types in C

Start recording!!

Chapter 16

Dynamic Memory Allocation

Dynamic Arrays

Say we need an array of Floats

- Do we know how big it will be ?
- Do we go for a ‘largest possible’ solution ?

NO !

- We waste memory
- Our code may crash as our ‘guess’ may be too small
- Passing the data in functions becomes time consuming

The Solution

Use a pointer and dynamically create the array as:

- We can ensure we have enough memory free
- We get exactly the right size array
- We can free the memory up when it is finished with
- We can pass all the data to analysis routines in one go
- We can even save all the data to file in one go!

Basically, we are writing good, robust, memory efficient code!

Pointers: in practice (1)

There are four ‘steps’ in using a pointer when using it to allocate an array

- Create the pointer
- Assign it a valid address in memory
- Use the memory associated with the pointer
- Free up the memory

There is the 4th step when using pointers with arrays (mentioned earlier!).

Pointers: in practice (2)

Create the pointer

- This is done the same as if we were declaring a pointer for a single variable

- It must be of the type we wish our array to be e.g.

```
int *a;          // For an array of integers

float *XData; // For an array of floats
```

Dynamic Memory Allocation

Allocate the pointer…

This is where the process differs.

- Previously we assigned the address of ***an existing variable*** to a pointer
- Now, we wish to ask for a base memory address where we can start to store values

There are two functions we can use for this

```
malloc

calloc
```

They are almost identical, often it is a mix of personal preference/code requirements that will determine which is used

malloc

Allocates a block of memory (given in bytes) **but does not initialise it**

```
void *malloc(size_t size);
```

Prototype in alloc.h and stdlib.h

Inputs:

- Size in bytes of memory requested to be allocated

Returns:

- a pointer to the Newly allocated block, or
- **NULL** if not enough space exists for the new block.

Note: If size == 0, it returns **NULL**.

calloc

Allocates space for n items of size bytes each and **initialises each item to zero**

```
void *calloc(size_t nitems, size_t size);
```

Prototypes in stdlib.h and alloc.h

Inputs:

- nitems: number of items to allocate memory for
- size: size, in bytes, of each item

Returns

- a pointer to the newly allocated block or
- **NULL** if not enough space exists.

free

Frees blocks allocated with

- `malloc` or
- `calloc`

Prototype is

```
void free (void *block);
```

Found in `stdlib.h` & `alloc.h`

Note:

- We must use this to return memory, it is NOT automatically done when a function exits (only the pointer is released)

An Example (or 3)

We will create dynamically an integer array of size ‘n’

- Fill it with the values 0 .. n
- Display the values (backwards)

Free up the memory used

LC16\dynamic_1.c , LC16\dynamic_2.c , LC16\dynamic_3.c ,

A note on multidimensional arrays

Just as we can allocate memory for a 1D array, we can do the same for 2D, 3D.. arrays.

- It is however somewhat complex
  - It involves creating arrays of pointers
  - Even more asterisks

- As such, we will not cover it now

Chapter 17

Function Programming (Part 3)

Passing arrays to functions (1)

In…

- Chapter 15: We saw how we can use a pointer to access items of an existing array
- Chapter 16: We developed the skills to allocate memory for an array (obtaining the base address to use)

Now

- Let us combine this with the knowledge on passing pointers to functions (Chapter 14)

Passing arrays to functions (2)

We can pass an entire array to a function simply by passing its address

- In fact, this is what C does for us ☺ (try and stop it!).

In the function call, we just provide the array variable,

- e.g. if we had declared an array as `int MyArray[1000];`

Our function call would be of the form

```
MyFunction ( MyArray );
```

Note: we could use: `MyFunction ( &MyArray[0] )` if we really wanted to!

Passing pointers to arrays to functions

When defining the function, we need to declare the parameter appropriately

Assuming we are passing an array of integers, the ‘easiest’ (and most obvious when reading code) is to declare the parameter as

`int ArrayToReceive[]` (the [] indicates an array without ‘fixing’ the size)

```
void MyFunction( int ArrayToReceive[] );
```

We could of course treat it as a pointer as it will be being passed a memory address, e.g.

```
void MyFunction( int *ArrayToReceive );
```

Does the same thing but not so obvious that it’s expecting an array

Using arrays passed to functions (1)

In the function we access the array (using any of the methods from chapter 15), e.g.

```
ArrayToReceive[n]
*(ArrayToReceive+n)
```

Note:

- As a pointer has been passed, the **original** array is accessed
- **NOT A COPY**
- i.e. you can (say) populate an array in a function
- The function does not know how big the array is! (Be careful!)

Using arrays passed to functions (2)

We Will create an integer array of size ‘n’

- Pass the array to a function (very efficiently!)
- Populate the array
- Pass the array to another function
- Display a result

Then repeat with a dynamically allocated array.

In this case we must free up the memory used at the end

LC16\DynamicFunction.c

Chapter 19

Advanced Data Types in C - Structures

Structures

Collections of variables stored under one name

Usually a set of related information

e.g. Name and address
Material properties
A set of coordinates

Defining a Structure (1)

keyword

structure tag

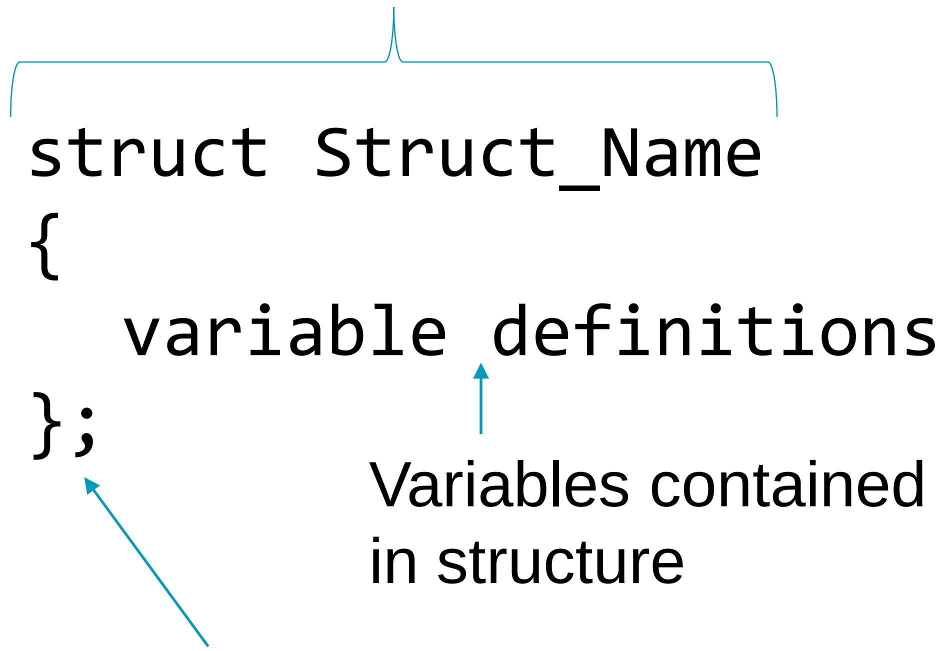
```
struct Struct_Name
{
   variable definitions
};
```

Defining a Structure (2)

structure type (used for variable declaration)

```
struct Struct_Name
{
  variable definitions
};
```

Defining a Structure (3)

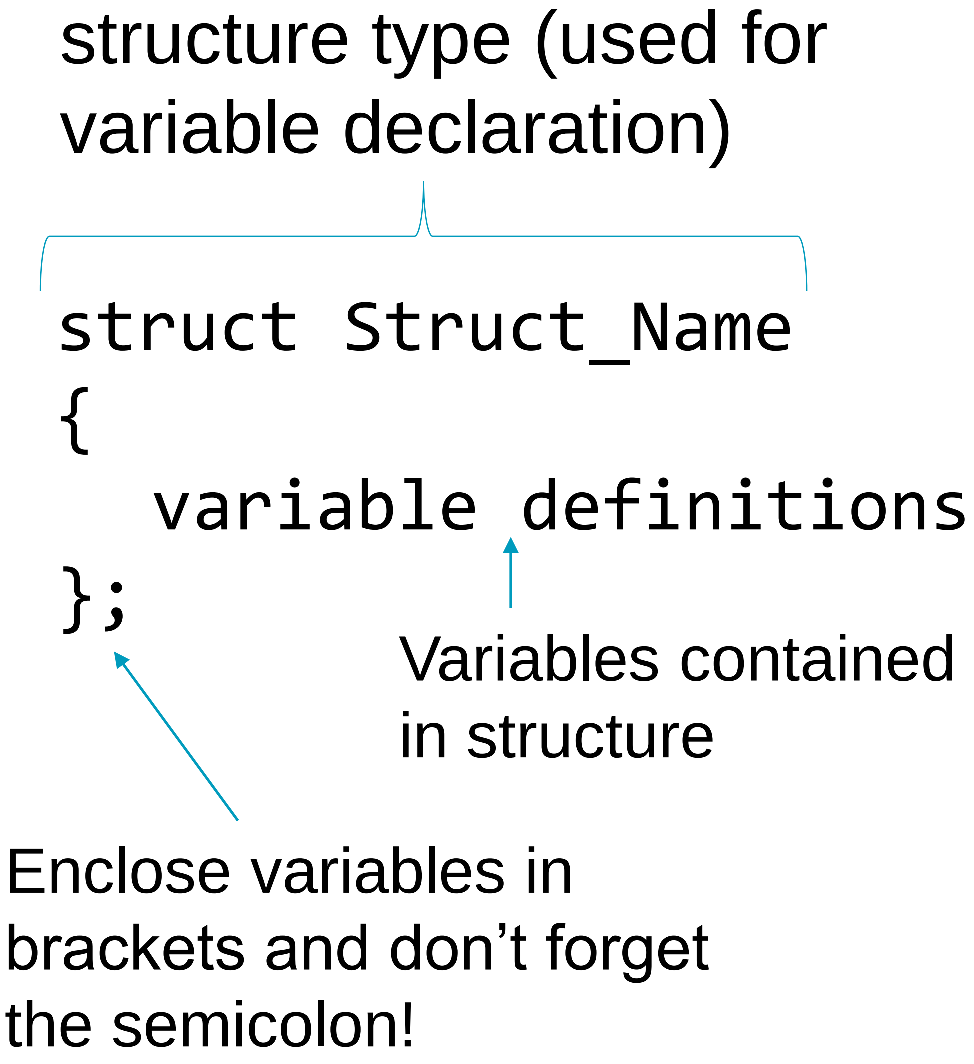


structure type (used for variable declaration)

```
struct Struct_Name
{
   variable definitions
};
```

Variables contained in structure

Enclose variables in brackets and don’t forget the semicolon!



structure type (used for variable declaration)

```
struct Struct_Name
{
   variable definitions
};
```

Variables contained in structure

Enclose variables in brackets and don’t forget the semicolon!

eg:

```
struct Employee
{
   int id, phone;
   char Surname[40];
   char Initials[5];
};
```

Structures

You can have any variable type within the structure definition

eg:

```
struct Employee
{
    int id, phone;
    char Surname[40];
    char Initials[5];
    float Salary[12];
    int Matrix[10][20];
    int *LookUp;
};
```

Using your Structure : Part 1

To declare a variable using the type of structure defined use the structure type specified (the keyword plus the tag)

```
struct Employee  JoeBlogs;
```

For an array of structures:

```
struct Employee  Employees[10];
```

For a pointer to a structure:

```
struct Employee  *Employee1;
```

Using your Structure : Part 2

Use the ‘dot operator’ to access individual structure variables:

```
  StructureName.Element_Name
```

For example:

```
   Name = Employees.Surname;
   Wage = Employees.Salary[0];
```

Using your Structure : Part 3

Individual Structures of the same type can be set equal to each other

eg.

```
      MyData[0] = MyData[4];
```

But **not**

```
      NewData = MyData;
```

LC19\structure_1.c , LC19\structure_2.c , LC19\structure_3.c

Structures and Functions

Structures can be passed to functions as a parameter

```
    MyFunc( employee1 )
```

Where the function declaration would be

```
    MyFunc( struct Employee employee )
```

Structure pointers

To change the data in the structure from within the function, pass a pointer

```
    MyFunc( &employee )
```

Then the function declaration would be

```
     MyFunc( struct Employee *employee )
```

Members of a structure can be accessed via the pointer using ->

```
    employee->Surname = “Smith”
```

Or

```
    (*employee).Surname = “Smith”
```

which allow members of a structure in the calling function to be accessed

LC19\StructureFunc.c

Chapter 19

Advanced Data Types in C – Enums, Const, Unions, #define and Advanced Structures

#define

When developing code, we aim to make it as readable and maintainable as possible.

One way is to define text labels (e.g. M_PI) that we can use in our code

We do this using the compiler directive #define, e.g.

```
#define UP 1
#define DOWN 2
```

Make sure you don’t put a semicolon at the end of the line – it will create problems when the compiler does the ‘find and replace’

#define ‘find and replace’

When we compile our code, the compiler does an initial ‘find and replace’ of these so (assuming #define UP 1) so,

```
if ( i == UP )     // easier for us to read
```

Becomes

```
if  ( i == 1)      // What is actually compiled
```

#define: A clever trick

```
#define Size 50
main()
{
   int Array[Size][Size];
   int iCols = Size;
   int iRows = Size;
```

While we do have to recompile code, this allows us to change the size of an array to match a problem.

If we use ‘Size’ in loops etc. we know too we will stay inside the array bounds.

This is also a good way for changing parameters within an application (e.g. you might #define a value for resistivity or permittivity which is then used in equations).

#define: A word of caution (1)

These need to be used with caution, one thing to note is **DO NOT** put a semicolon on the end of a #define e.g.

```
#define UP 1;
```

As

```
if ( i == UP )   // easier for us to read
```

Becomes

```
if  ( i == 1;)   // What is actually going to be compiled
```

Which is an error and does not compile!

#define: A word of caution (2)

As this is a simple ‘find and replace’ the compiler cannot spot the following problem (and is very hard for us to track down as it is not an ‘error’).

```
#define UP 1
#define DOWN 1

if ( i == UP )
{
    // some code for ‘UP’
}

if ( i == DOWN )
{
    // some code for DOWN
}
```

Written as

```
#define UP 1
#define DOWN 1

if ( i == 1 )
{
    // some code for ‘UP’
}

if ( i == 1 )
{
    // some code for DOWN
}
```

Compiled as

Enumerated Types

This is a simple way of defining an integer type and setting unique, incrementing values to them (in one go!)

```
      enum  Enum_Name  { types }
```

Eg

```
      enum Days { mon, tue, wed, thu, fri } ;
```

It also has the advantage that the numbers NEVER replicate – so avoiding the previous problem.

Starting values for enum’s

By default, enum’s start at zero

```
      enum Days { mon, tue, wed, thu, fri }
```

So mon = 0, tue = 1 etc

But we can define a start value

```
      enum Days { mon=1, tue, wed, thu, fri }
```

So mon = 1, tue = 2 etc

C19\enum.c TestEncoder

Static Variables

A static variable is one that when defined in a function is not destroyed when the function terminates.

It holds the value and can be accessed the next time the function is called.

Often used to count the number of times a function is called.

C19\static.c

const variables

`const` is a keyword used to make the value of an identifier constant.

```
    const int X = 30;
```

Unlike #define, **`const`** is scope controlled

A `const` variable will have memory space allocated to it (but this may be device/compiler dependent)

https://www.baldengineer.com/const-vs-define-when-do-you-them-and-why.html

LC19\ConstHashDefine.c

Unions

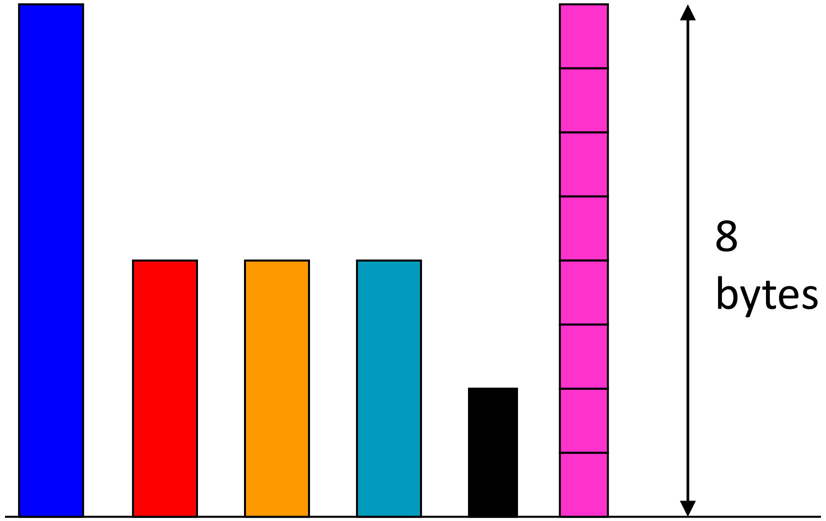
A union is a set of variables that

- Overlap
- Start at the same place in memory
- They are defined in a similar fashion to structures
- They can be used to save memory

Unions – a graphical explanation

The storage for each variable overlaps – in this case the char array is defined to cover the whole range of bytes of memory used.

```
union number
{
        double d;
        float f;
        long l;
        int i;
        short s;
        unsigned char c[8];
};
```



Note: sizes in bytes will be machine dependent

MotorControlSkeleton.ino

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.