CC2511 Week 3: Lecture 1

More on the C language

Reminder

- Submit your Assignment 1 CAD files to LearnJCU.

C language reference

- Continue on the C language reference (up to functions)

Functions in C

- Example:

```
double sum(double a, double b)
{
    return a + b;
}
```

- Return type: **double**. Function name: **sum**. Arguments: **a** and **b**, both of type **double**.
- The keyword **return** ends execution there and returns to the caller.

External functions

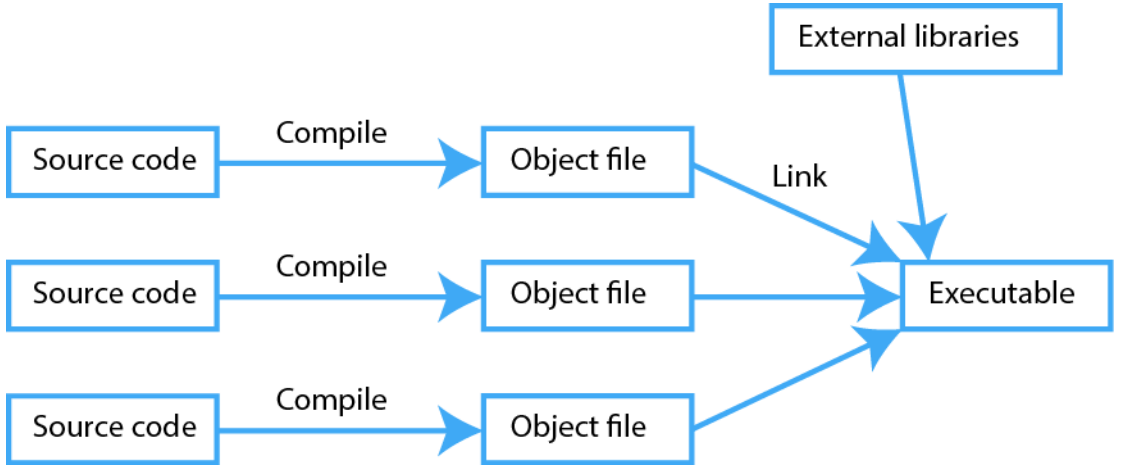
- Consider this code:

```
void func(int i)
{
    func2(i + 1);
}
```

- How does the compiler know that `func2` exists?
  - How does it know what data types its arguments are?

Linkage

- Each file is compiled separately.
- The **compiler** does not necessarily see the code for all the functions that are needed.
- The **linker** must find the code for all the relevant functions.



Function prototypes

- The compiler looks at only one C file at a time, but the source code may call functions defined in other files.
- The compiler needs to know the number of arguments and their data types in order to perform the static type checking.
  - e.g. if the function asks for a char but you pass an int there will be an error.
- Functions and their data types need to be **declared ("prototyped")** before they can be used.

Function prototypes

- A function prototype looks like:

```
return_type function_name(type arg1, ...);
```

- e.g.:

```
int func(int i);
int func(int i, char c, double x);
```

Header files

- By convention, functions defined in file.c have their prototypes in file.h
- The “.h” extension stands for “header”
- Header files are “included” into C files in order to prototype the necessary functions.

The C pre-processor

- Syntax for including a header file:

  #include <stdio.h>

- This is a **pre-processor** directive that runs before the main compiler.
- #include means to **take the contents of the named file and insert it into the current one**.
- All pre-processor directives have # in the first column of the line

Pointers

- **Pointers** are a new data type that contain a **memory address**.
- In contrast with standard variables that contain a value
  - e.g. an “int” type contains an integer.
- A pointer to an int, “**int ***” contains an address which is understood to point to an integer.

Examples of declaring pointers

```
int *ptr_to_int;
bool *ptr_to_bool;
double *ptr_to_double;
```

- Notice the unusual placement of the space and the asterisk *

Why the space-then-asterisk form?

- **The C language has an unusual syntax rule here. Pay attention**.

```
int* a, b, c;
// a is a pointer but b and c are NOT pointers!

// If we wanted them all to be pointers, write:
int *a, *b, *c;
```

Assigning pointers

- Pointers can be assigned to just like any variable.

```
byte *p;
p = 0x1234;
// p now points to memory address 1234 (hexadecimal)
```

Assigning pointers: a real example

GPIO memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
400F_F000	Port Data Output Register (GPIOA_PDOR)	32	R/W	0000_0000h	47.2.1/ 1182

```
int *GPIOA_PDOR = 0x400FF000;

// The pointer GPIOA_PDOR now points to the
// Port Data Output Register for GPIO port A
```

The address-of operator

- We can generate a pointer to any variable using the address-of operator &

```
byte val;
byte *p = &val;
// Notice the & before val
// The pointer p now points to val
```

Dereferencing pointers

- Recall that

```
int *GPIOA_PDOR;
GPIOA_PDOR = 0x400FF000;
```

- assigns to the pointer and not the address it points to.

- Using the address that a pointer points to is called **dereferencing** the pointer.

Dereferencing pointers

- Dereference pointers using the * operator

```
int *GPIOA_PDOR;

// assigns to the pointer:
GPIOA_PDOR = 0x400FF000;

// assigns to the address pointed to by the pointer:
*GPIOA_PDOR = 0x22; // notice the leading *
```

Arrays

- An **array** is a numbered, ordered collection of values of the same data type.
- Declaring arrays:

```
int array [10];
```

- This declares a variable `array` to hold 10 integers.

Subscripting arrays

- To access elements in arrays use square brackets:

```
int arr [10];
arr[0] = 1; // notice indices start from zero
arr[1] = 10;
```

How arrays are implemented

- Consider an **8 bit microcontroller** (simply to make the addresses shorter to write).
- Suppose that data memory begins at address 0x10 on this machine.

Variable	Address	Value
	0x10	?
	0x11	?
	0x12	?
	0x13	?
	0x14	?
	0x15	?
	0x16	?
	0x17	?
	0x18	?
	…	

Arrays are contiguous in memory

```
char a = 0;
char b = 1;
char array [4];
```

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	?
	0x13	?
	0x14	?
	0x15	?
	0x16	?
	0x17	?
	0x18	?
	…	

Index arrays using square brackets starting from zero

```
char a = 0;
char b = 1;
char array [4];

// The first item is [0]
array[0] = 100;
array[1] = 101;
```

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	100
	0x13	101
	0x14	?
	0x15	?
	0x16	?
	0x17	?
	0x18	?
	…	

Size of the array

```
char a = 0; char b = 1;
char array [4];
char c = 2;

array[0] = 100;
array[1] = 101;
```

- What is `array[4]`?

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	100
	0x13	101
	0x14	?
	0x15	?
c	0x16	2
	0x17	?
	0x18	?
	…	

Size of the array

- **Accessing memory beyond the end of an array is not a compile error in C!**
- You can write

```
array[4] = 200;
```

and you will modify some other (unintended) memory!

- (It is not guaranteed to be the next variable in your list of declarations.)

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	100
	0x13	101
	0x14	?
	0x15	?
c	0x16	200
	0x17	?
	0x18	?
	…	

Buffer overflows

- Reading or writing beyond the end of an array is called a “buffer overflow”.
- It is probably the single most common fault in the history of computing.
- The fact that the compiler doesn’t protect against this has been called “C’s biggest mistake”.

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	100
	0x13	101
	0x14	?
	0x15	?
c	0x16	200
	0x17	?
	0x18	?
	…	

Arrays are pointers

```
char a = 0;
char b = 1;
char array [4] = {100,200,300,400};
```

- The name of the array is a pointer to the first item.

```
int x = array; // contains 0x12
int y = array[0]; // contains 100
```

Variable	Address	Value
a	0x10	0
b	0x11	1
array	0x12	100
	0x13	200
	0x14	300
	0x15	400
	0x16	?
	0x17	?
	0x18	?
	…	

Array indexing is pointer arithmetic

- These two expressions are equivalent:

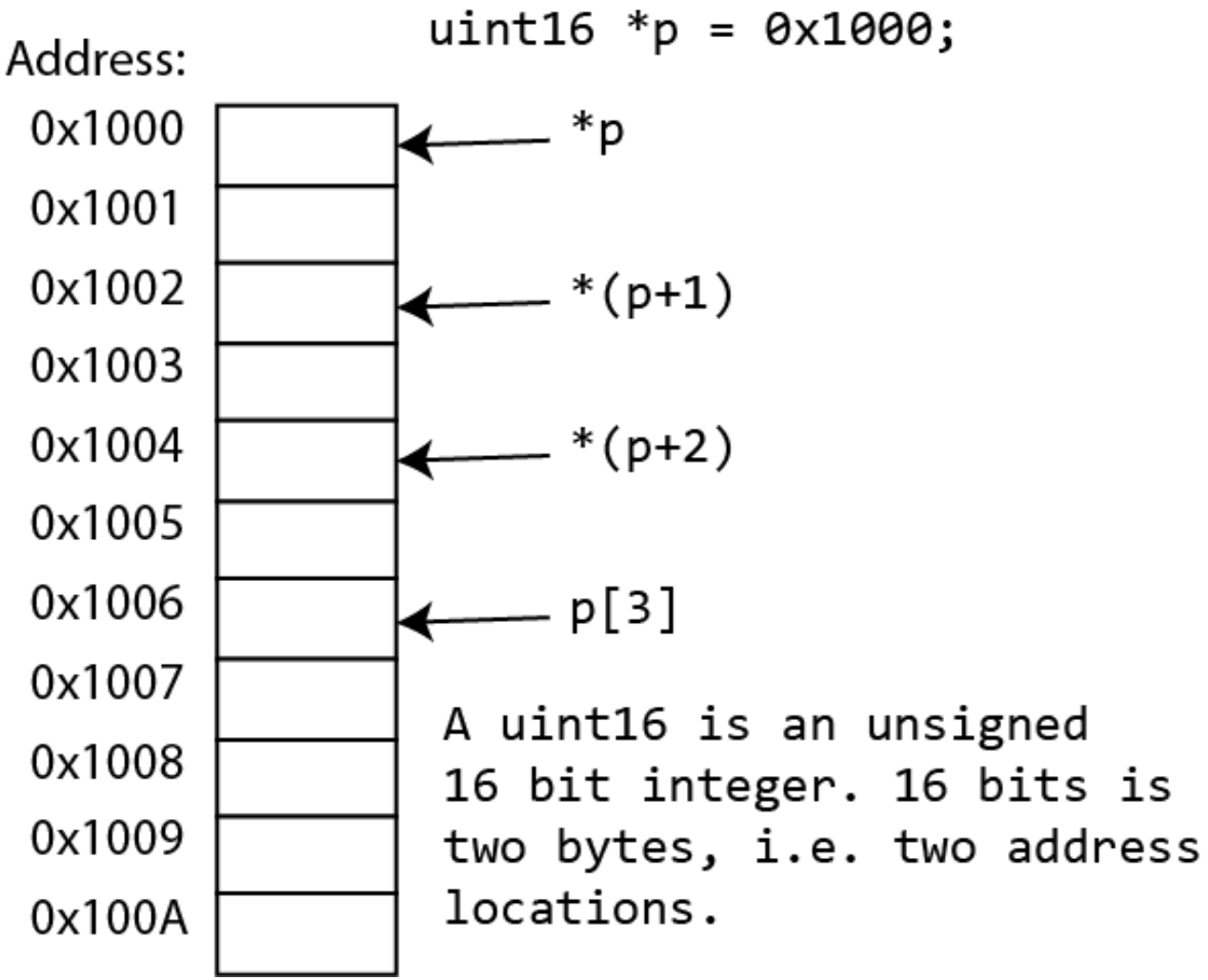
```
myarray[3]
*(myarray + 3)
```

- The square brackets is simply “syntactic sugar” for pointer arithmetic.

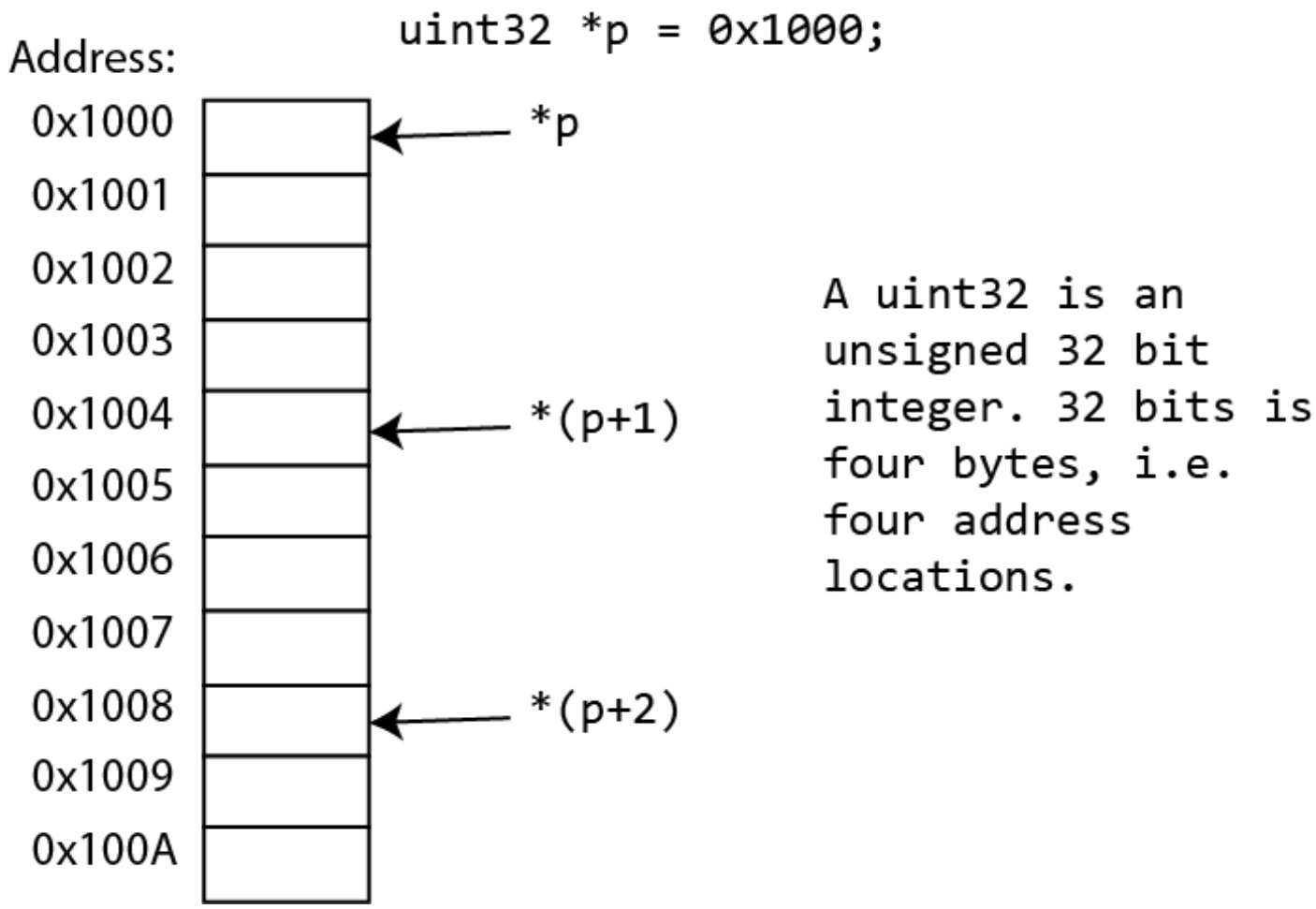
Pointer arithmetic

- The C compiler knows the size of each data type.
- Pointer arithmetic moves in units of the size of the data type.

Pointer arithmetic



Pointer arithmetic



A uint32 is an unsigned 32 bit integer. 32 bits is four bytes, i.e. four address locations.

What would this code do?

- What would this code do?

```
int values[5];
*(values + 3) = 8;
```

What would this code do?

- What would this code do?

```
int values[5];
*(values + 3) = 8;
```

- Remember this is just arithmetic!

```
    *(values + 3) = 8;
```

is the same as

```
    values[3] = 8;
```

C reference

- Continue working on the C syntax reference