

Structures

COMP2017/COMP9017

Dr. John Stavrakakis

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



What is a *Structure*?

- › So far the only collection of data we've covered is the *array*
 - › Arrays are used to hold items of the **same type** and access them by giving an index
 - › Sometimes we want to hold a collection of data items of ***different*** types.
 - › For example: a library catalogue for a book might contain the title, author's name, call number, date acquired, date due back etc
 - › For this type of collection C has a data type called a ***structure***
-

Structure definition example

struct date

{

enum day_name day;

int day_num;

enum month_name month;

int year;

};

name of the type of structure

fields of the structure

structure example

```
struct date {  
    enum day_name    day;  
    int              day_num;  
    enum month_name  month;  
    int              year;
```

```
} Big_day    {  
    Mon, 7, Jan, 1980  
};
```

```
struct date    moonlanding;  
struct date    deadline = {day_undef, 1, Jan,  
                           2000};  
struct date    *completion;
```



```
struct date {  
    enum day_name    day;  
    int              day_num;  
    enum month_name  month;  
    int              year;
```

Structure definition

```
} Big_day
```

Structure declaration

```
{
```

```
    Mon, 7, Jan, 1980
```

Structure initialisation

```
};
```

```
struct date    moonlanding;  
struct date    deadline = {day_undef, 1, Jan, 2000};  
struct date    *completion;
```

```
struct date {
    enum day_name    day;
    int              day_num;
    enum month_name   month;
    int              year;
};

struct date  moonlanding;

struct date  deadline = {day_undef, 1, Jan, 2000};

struct date  *completion;
```



```
struct car_desc
{
    enum car_cols    colour;
    enum car_make    make;
    int              year;
};
```



```
struct [tag]  
{  
    member-declarations  
  
} [identifier-list];
```

- › Once tag is defined, can declare structs with:

```
struct tag identifier-list;
```



Accessing Elements of a struct

```
struct date bigday;
```

```
int         theyear;
```

```
theyear = bigday.year
```

A dot used to nominate an element of the structure.

Accessing Elements of a struct

```
struct date bigday;
```

```
struct date * mydate;
```

```
int    theyear;
```

```
mydate = &bigday;
```

If a pointer to the structure is used, then the -> operator indicates the element required.

```
theyear = mydate->year
```



typedef

```
typedef struct date{  
    enum day_name    day;  
    int              day_num;  
    enum month_name  month;  
    int              year;  
} Date;
```



typedef

```
typedef struct date{  
    enum day_name    day;  
    int              day_num;  
    enum month_name  month;  
    int              year;  
} Date;
```



```
typedef struct date{
```

```
    enum day_name      day;  
    int                day_num;  
    enum month_name    month;  
    int                year;
```

```
} Date;
```

```
Date Big_day = {Mon, 7, Jan, 1980};
```

```
Date moonlanding;
```

```
Date dopday = {day_undef, 1, Jan, 2000};
```

```
Date *completion;
```

Struct: function arguments, returns

```
struct customer    s1;
struct salesrep    s2;
struct sale transact(struct customer s1, struct salesrep s2);

struct sale transact(struct customer s1,
                    struct salesrep s2)
{
    struct sale  sl;
    ...
    return sl;
}
```



- > `stdio.h`
 - > `time.h`
 - > `stat.h`
 - > `pwd.h`
-

```

struct tm
{
    int  tm_sec; /* Seconds.    [0-60]  */
    int  tm_min; /* Minutes.  [0-59]  */
    int  tm_hour; /* Hours.    [0-23]  */
    int  tm_mday; /* Day.      [1-31]  */
    int  tm_mon;  /* Month.    [0-11]  */
    int  tm_year; /* Year - 1900. */
    int  tm_wday; /* Day of week. [0-6]  */
    int  tm_yday; /* Days in year. [0-365] */
    int  tm_isdst; /* DST indicator */
    long int tm_gmtoff; /* Seconds east of UTC. */
    const char *tm_zone; /* Timezone abbreviation. */
};

```

```

struct tm * localtime(long *); /* forward decl. */
struct tm * now;

```

```

now = localtime(&sometime);
    /* sometime contains time in seconds after
    Jan 1 1970 */

```

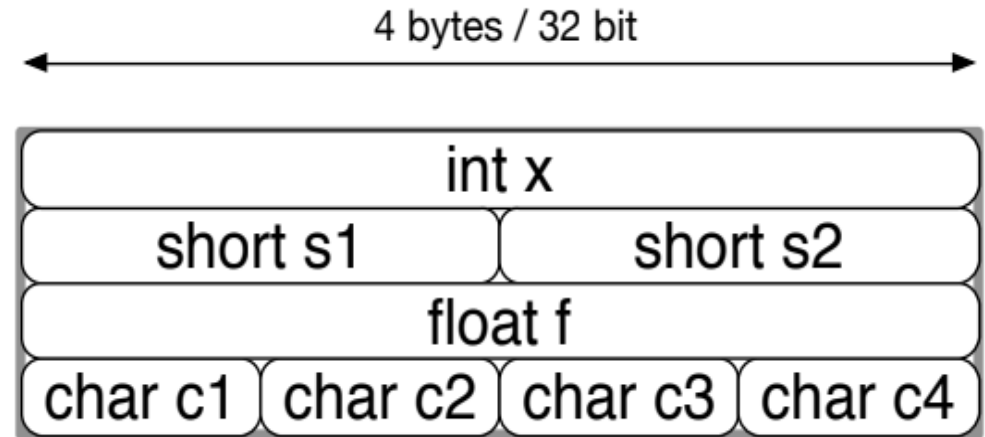



```
Hour_now = now->tm_hour;
```

```
printf ("%d/%d/%d\n", now->tm_mday, now->tm_mon,  
        now->tm_year);
```

Memory alignment

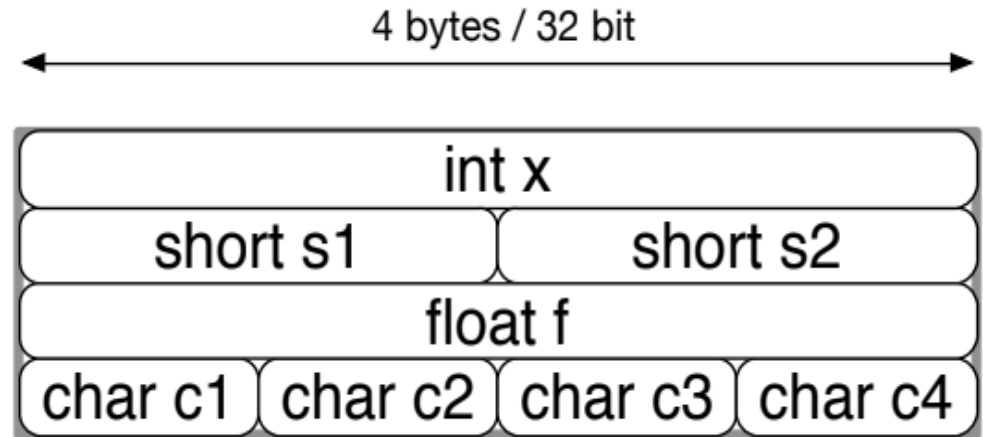
```
struct a {  
    int x;  
    short s1, s2;  
    float y;  
    char c1, c2, c3, c4;  
};
```



```
sizeof (struct a) == 16
```

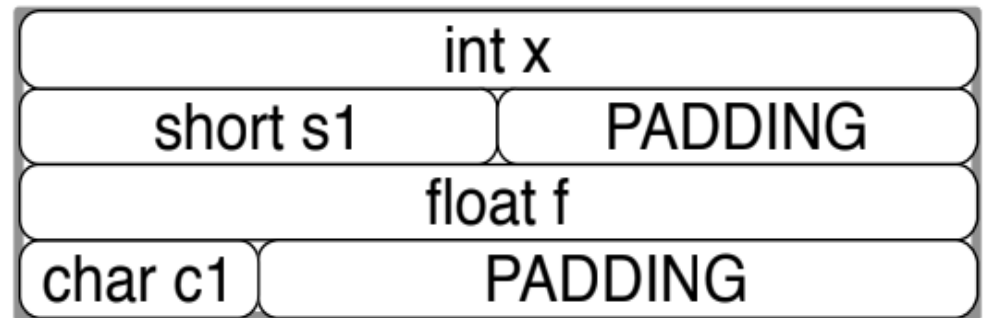
Memory alignment

```
struct a {  
    int x;  
    short s1, s2;  
    float y;  
    char c1, c2, c3, c4;  
};
```



`sizeof (struct a) == 16`

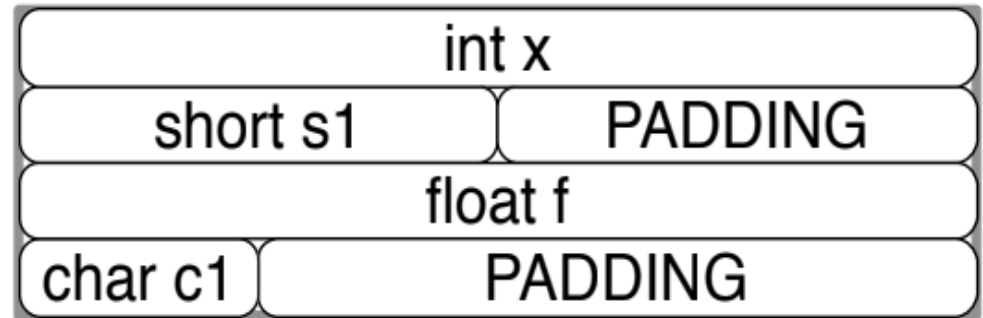
```
struct b {  
    int x;  
    short s1;  
    float y;  
    char c1;  
};
```



`sizeof (struct b) == 16`

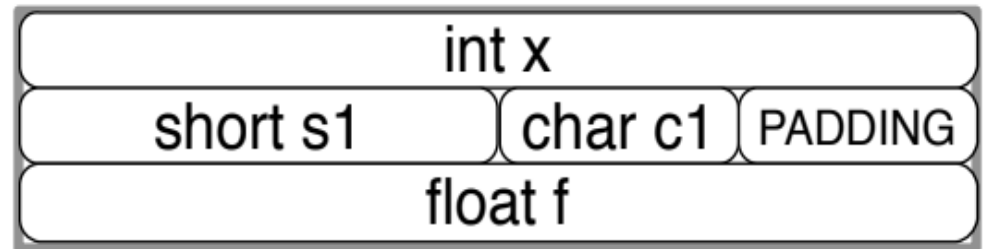
Memory alignment

```
struct b {  
    int x;  
    short s1;  
    float y;  
    char c1;  
};
```



`sizeof (struct b) == 16`

```
struct c {  
    int x;  
    short s1;  
    char c1;  
    float y;  
};
```

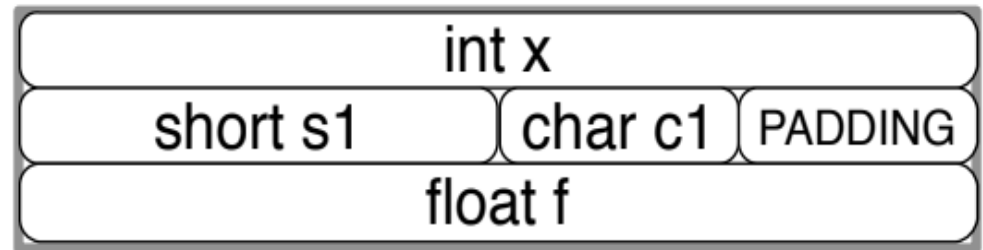


`sizeof (struct c) == 12`

Memory alignment

- Address of a struct variable will give us direct access to bytes of the first members
 - Alignment depends on architecture
 - Special compiler extensions can be used to prevent padding
 - h/w speed/memory

```
struct c {  
    int x;  
    short s1;  
    char c1;  
    float y;  
};
```



```
sizeof (struct c) == 12
```

Unions

COMP2017/COMP9017

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY





- › Sometimes we want several variants of a structure but don't want to consume more memory
- › the C *union* lets you declare variables that occupy the **same** memory

- › A library catalogue that contains information about books and films
- › for books we want to store:
 - author
 - ISBN
- › for films we want to store:
 - director
 - producer


```
enum holding_type {book, film};
struct catalog
{
    char * title;
    enum holding_type type;
    struct /* book */
    {
        char * author;
        char * ISBN;
    } book_info;
    struct /* film */
    {
        char * director;
        char * producer;
    } film_info;
};
```

Solution 1

How many bytes
total?

only one of the
structures

book_info or
film_info is used
at any one time.

this can be a major
waste of memory

- › in the first solution, only one of the structures `book_info` or `film_info` is used at any one time.
- › this can be a major **waste of memory**
- › instead, we can use a *union* to indicate that each variant occupies the **same** memory area

```
enum holding_type {book, film};
struct catalog
{
    char * title;
    enum holding_type type;
    union
    {
        struct /* book */
        {
            char * author;
            char * ISBN;
        } book_info;

        struct /* film */
        {
            char * director;
            char * producer;
        } film_info;
    } info;
};
```

Solution 2

we can use a *union* to indicate that each variant occupies the **same** memory area

> to access elements of a union we use the notation
`union_name.part_name`

> example:

\equiv int \equiv
 \equiv char \equiv

union

{

int a;

char b;

} x;

11	22	33	44
----	----	----	----

x.a = 0x11223344;

> to access elements of a union we use the notation `union_name.part_name`

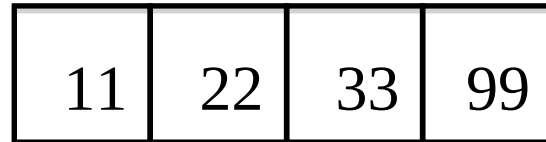
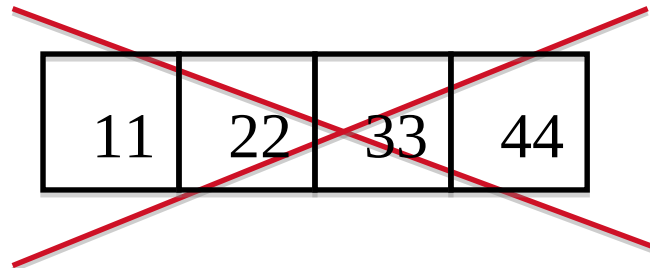
> example:

union

```
{
    int    a;
    char   b;
} x;
```

≡ int ≡

≡ char ≡



x.a = 0x11223344;

x.b = 'c';

- › in our example, we would access the author this way:

```
struct catalogx;
```

```
x.info.book_info.author
```



- › How can you tell what variant of the union is being used?
- › Answer: you can't!
- › need to have a separate variable to indicate variant in use

Access Example

```
struct catalog x;
```

an enum that indicates the variant

```
switch (x.holding_type)
```

```
{
```

```
    case book:
```

```
        printf("author: %s\n", x.info.book_info.author);
```

```
        break;
```

```
    case film:
```

```
        printf("producer: %s\n", x.info.film_info.producer);
```

```
        break;
```

```
}
```


Bitfields

COMP2017/COMP9017

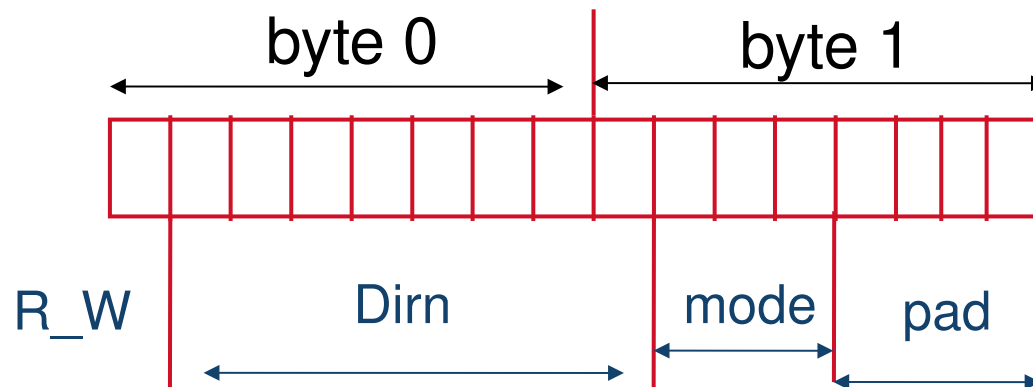
FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



- > for some specialised applications you need data fields that are smaller than a byte or are packed into several bytes



- › can specify a size, in bits, for elements of a structure
- › the size is placed after the field name, with a colon between:

```
struct IOdev  
{  
    unsigned R_W: 1;  
    unsigned Dirn: 8;  
    unsigned mode: 3;  
};
```

**this variable occupies
only 3 bits**



```
struct IOdev
```

```
{
```

```
    unsigned R_W: 1;
```

```
    unsigned Dirn: 8;
```

```
    unsigned mode: 3;
```

```
    unsigned pad: 4;
```

```
};
```

```
struct IOdev    dev = {1, 0, 7};
```

```
void main()
```

```
{
```

```
    printf("mode = %d\n", dev.mode);
```

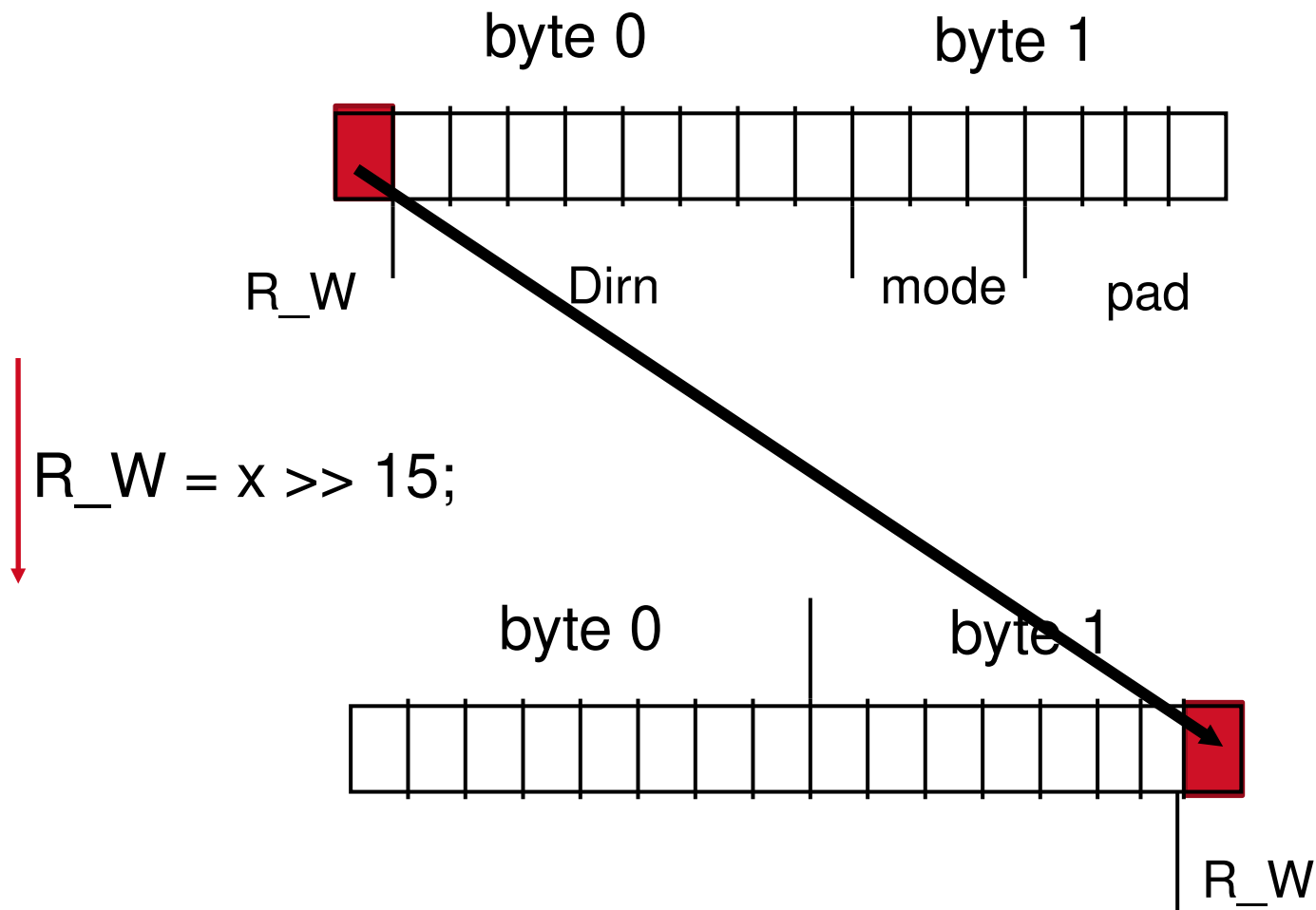
```
}
```

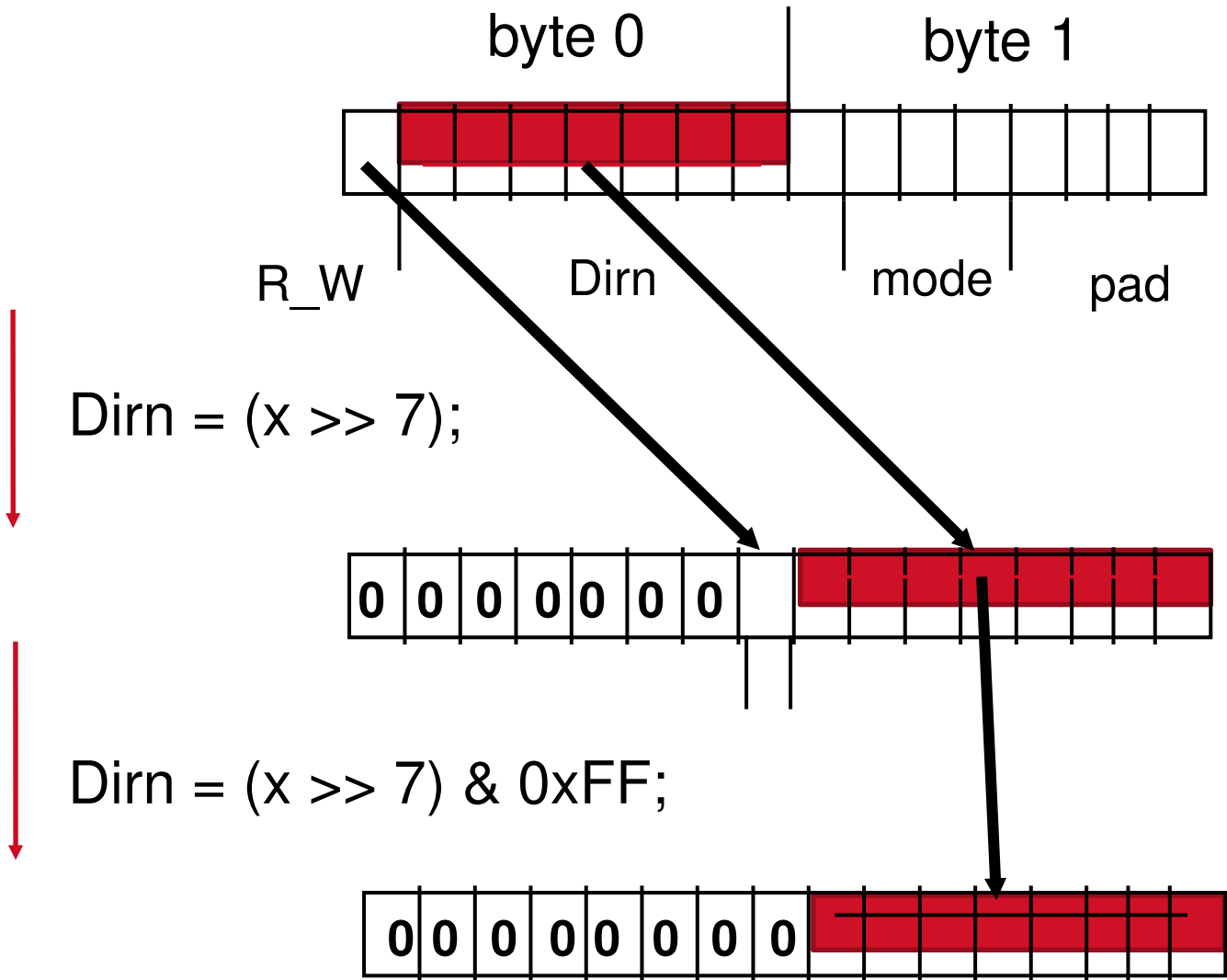
- › bitfields are good for low level programming of device registers (drivers, embedded systems etc)
- › bitfields are good for “unpacking” data structures
- › **however** bitfields may not be portable
 - padding
 - left-right vs right-left
- › only for experts!

- › without using the C bitfield syntax you can still unpack bit fields from data
- › use shift and logical operations
- › eg assuming previous packing of R_W etc:

```
unsigned short x; /* R_W:1, Dirn:8, mode:3, pad:4  
*/
```

```
R_W = x >> 15;  
Dirn = (x >> 7) & 0xFF;  
mode = (x >> 4) & 0x7;
```





- › shift right: \gg
- › shift left: \ll
- › bitwise AND: $\&$
- › bitwise OR: $|$
- › bitwise XOR: \wedge
- › bitwise NOT: \sim
 - Not to be confused with logical NOT !



- › bitfields: easy packing/unpacking of short bit fields
- › bit operations: shifting and logical

Files in C

COMP2017/COMP9017

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



What are files?

- › Disk storage peripherals provide persistent storage with a low-level interface
 - Fixed-size blocks
 - Numeric addresses
 - › Operating system arranges this into an **abstraction as files**
 - Files can be variable length
 - Files have names, meta-data (owner, last modified date, etc)
 - Files are arranged into eg a tree, by folder/directory structure
 - › Read or write a file is done through System Calls (APIs)
-

- › Devices are often represented as files
 - software reads/writes file to access the device
 - E.g. Send a command to the printer by writing to a particular file name
 - › If a file can be a physical device, then it is not fixed in size or behaviour.
 - › A *stream* is associated with a file
 - May support a file position indicator [0, file length] *
 - Can be binary or not (e.g. ASCII, multibyte)
 - Can be open/closed/flushed!
 - Can be *unbuffered*, *fully buffered* or *line buffered*
-

- › For each file opened, there needs to be a file descriptor
 - › The descriptor describes the state of the file
 - Opened, closed, position etc.
 - › **#include <stdio.h>**
 - contains many standard I/O functions and definitions for using files
-

- > **FILE** is a struct that is defined in `stdio.h` and this is the descriptor
- > To open a file, we use the **fopen** function

FILE *fopen(const char *path, const char *mode);

FILE * myfile = fopen("turtles.txt", "w");

variable

filename

mode

> **FILE *fopen(...)**

- modes

r open text file for reading
w truncate to zero length or create text file for writing
a append; open or create text file for writing at end-of-file
rb open binary file for reading
wb truncate to zero length or create binary file for writing
ab append; open or create binary file for writing at end-of-file
r+ open text file for update (reading and writing)
w+ truncate to zero length or create text file for update
a+ append; open or create text file for update, writing at end-of-file

> File versions of your lovable input/output

- **fscanf**

- **fprintf**

Binary data use

- **fread**
- **fwrite**

> Finish off with **fclose**

- › When your program begins, special files are opened for you:
 - `stdin`, `stdout`, `stderr`
- › You can use these files
 - `fscanf(stdin, ...)` same as `scanf(...)`
 - `fprintf(stdout, ...)` same as `printf(...)`
- › When a stream supports file position, the position is zero
 - Every print/scan operation adjusts the position in the stream
 - Query position `ftell`, change position `fseek`

- › For reading input files, e.g. **stdin**, the end of file is important
 - **feof()** tests the end of file indicator
 - EOF does not happen until trying to read beyond end of stream

```
while ( ! feof(stdin) ) {  
    int num;  
    fscanf(stdin, "%d", &num);  
    fprintf(stdout, "num: %d\n", num);  
}
```

```
$ ./printnum < twonum.txt
```

- › For reading input files, e.g. **stdin**, the end of file is important
 - **feof()** tests the end of file indicator
 - EOF does not happen until trying to read beyond end of stream

```
while ( ! feof(stdin) ) {  
    int num;  
    fscanf(stdin, "%d", &num);  
    fprintf(stderr, "num: %d\n", num);  
}
```

```
while ( ! feof(stdin) ) {  
    int num;  
    int nread = fscanf(stdin, "%d", &num);  
    if (nread <= 0)  
        break;  
    fprintf(stdout, "num: %d\n", num);  
}
```

- › unbuffered – input/output is passed on as soon as possible
 - › fully buffered – input/output is accumulated into a block then passed
 - › line buffered – the block size is based on the newline character
 - › Which do you get? Depends.
 - Device driver writers should consider **setvbuf** for optimal block size
 - › **fflush**
 - Output stream: force write all data,
 - Input stream: discard any unprocessed buffered data.
-

- › Many problems with **fscanf** with rules about whitespace, newlines or complex format string
 - › **fgets** reads **one line** of input and returning a string (with the newline character)
 - Use string processing functions to deal with the returned data
 - › Use **fgets** correctly, together with **feof** to distinguish read errors vs end of file.
 - it will make life easier
 - › **ferror** when you get that feeling...
-

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

#define BUFLLEN (64)

int main(int argc, char **argv) {
    int len;
    char buf[BUFLLEN];
    while (fgets(buf, BUFLLEN, stdin) != NULL) {
        len = strlen(buf);
        printf("%d\n", len);
    }
    return 0;
}
```
