Structures

COMP2017/COMP9017

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

name of the type of structure

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

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
};
```

- placeholder in memory
- define a new variable, and automatically renew contents
- a pointer that points to address which is large enough to store struct data type

```
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;

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

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

Structure definition

Structure declaration

Structure initialisation

Structures

```
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;
```

Structures

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

General form

```
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;

   (*mydate).year
  - type casting from address to struct date

theyear = mydate->year
```

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

typedef

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

typedef alians struct date to date

typedef

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

but typedef will hide some information

```
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  s1;
        ...
        return s1;
}
```

all the memory associate
with custromer structure is
copy to the function

Standard structures

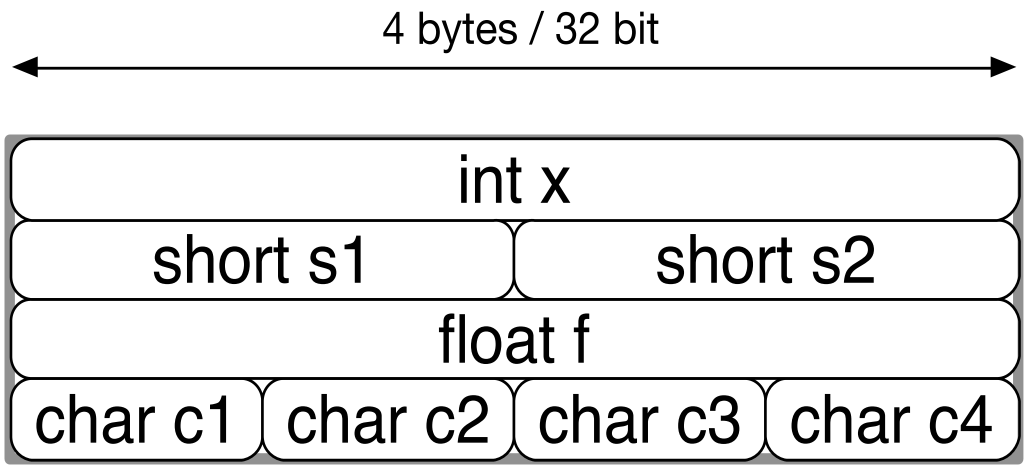
- `stdio.h`
- `time.h`
- `stat.h`
- `pwd.h`

```
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 */
```

```c
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;
};
```

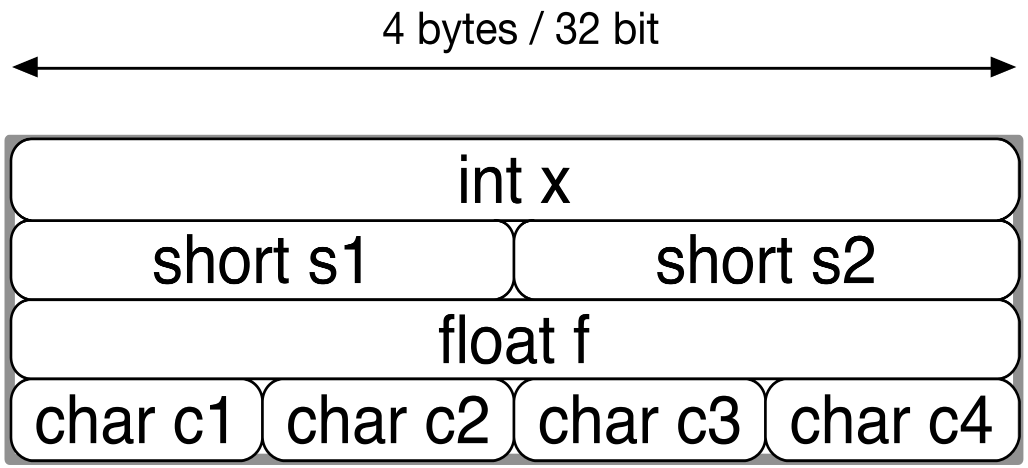



```
sizeof (struct a) == 16
```

THE UNIVERSITY OF SYDNEY

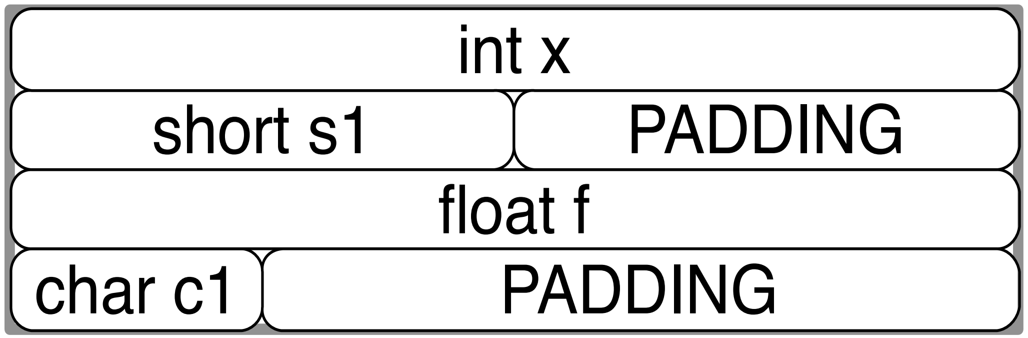
Memory alignment

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




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

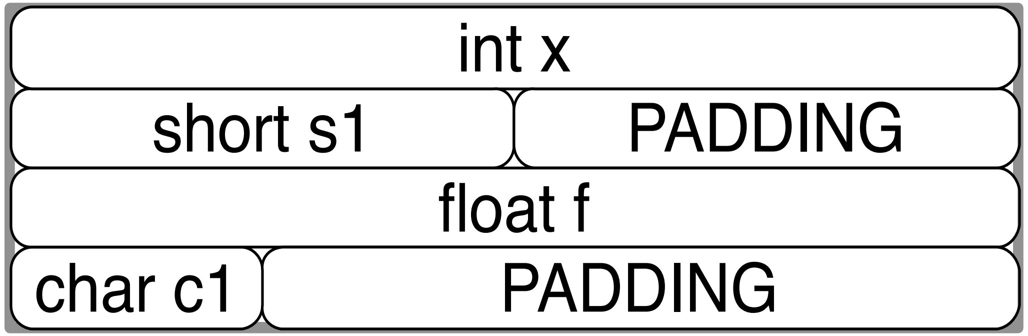
```
struct b {
    int x;
    short s1;
    float y;
    char c1;
};
```




```
sizeof (struct b) == 16
```

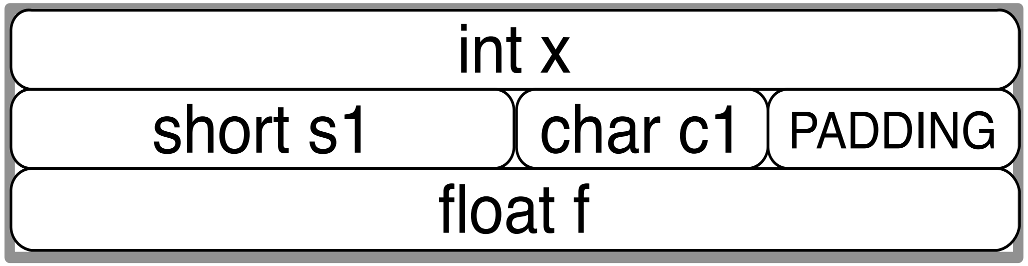
Memory alignment

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




```
sizeof (struct b) == 16
```

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

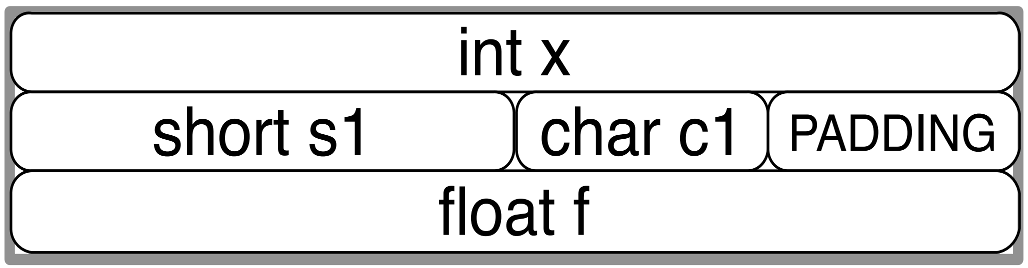



```
sizeof (struct c) == 12
```

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

Unions

- 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

Example

- 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

```c
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

First Solution

- 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

```c
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

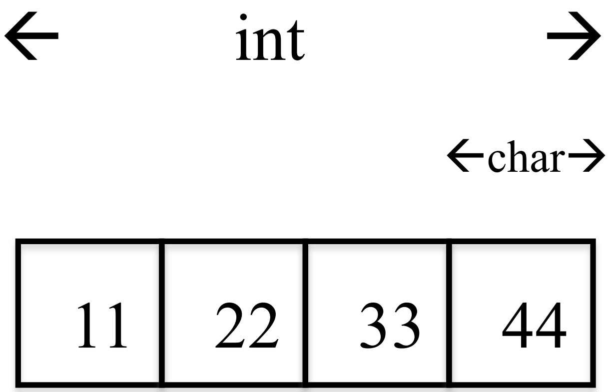
we can use a *union* to indicate that each variant occupies the **same** memory area

Access

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

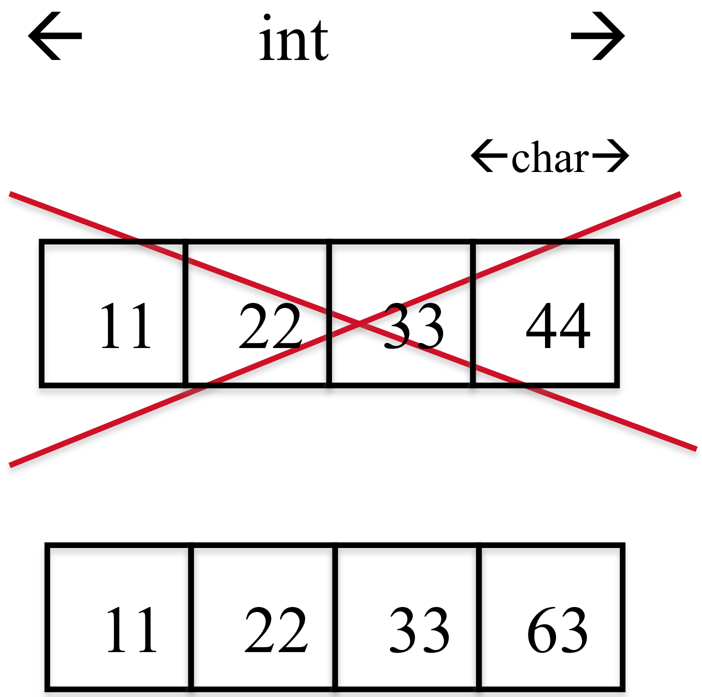
› example:

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

x.a = 0x11223344;
```



- to access elements of a union we use the notation `union_name.part_name`
- example:

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




```
x.a = 0x11223344;
x.b = 'c';
```

Access

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

```
struct catalog x;

x.info.book_info.author
```

Access

› 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

an enum that indicates the variant

```
struct catalog x;

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

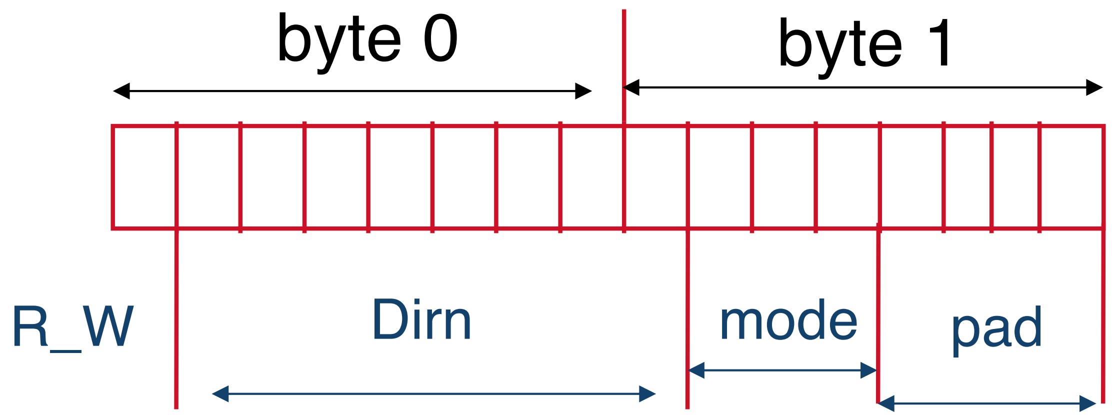
COMP2017/COMP9017

FACULTY OF ENGINEERING

THE UNIVERSITY OF SYDNEY

Bit fields

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



Bit Fields

- 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);
}
```

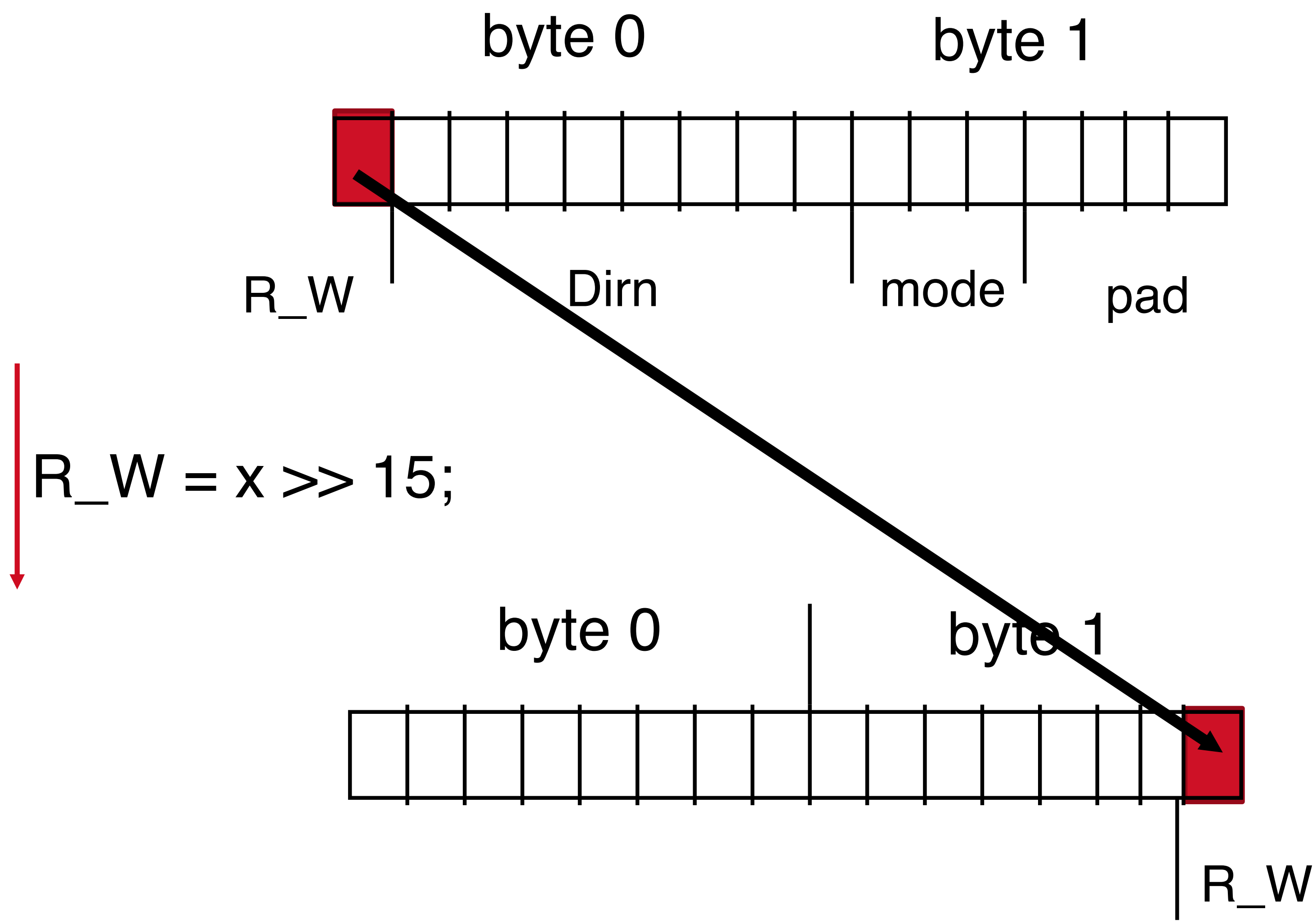
Bit Fields

- 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!

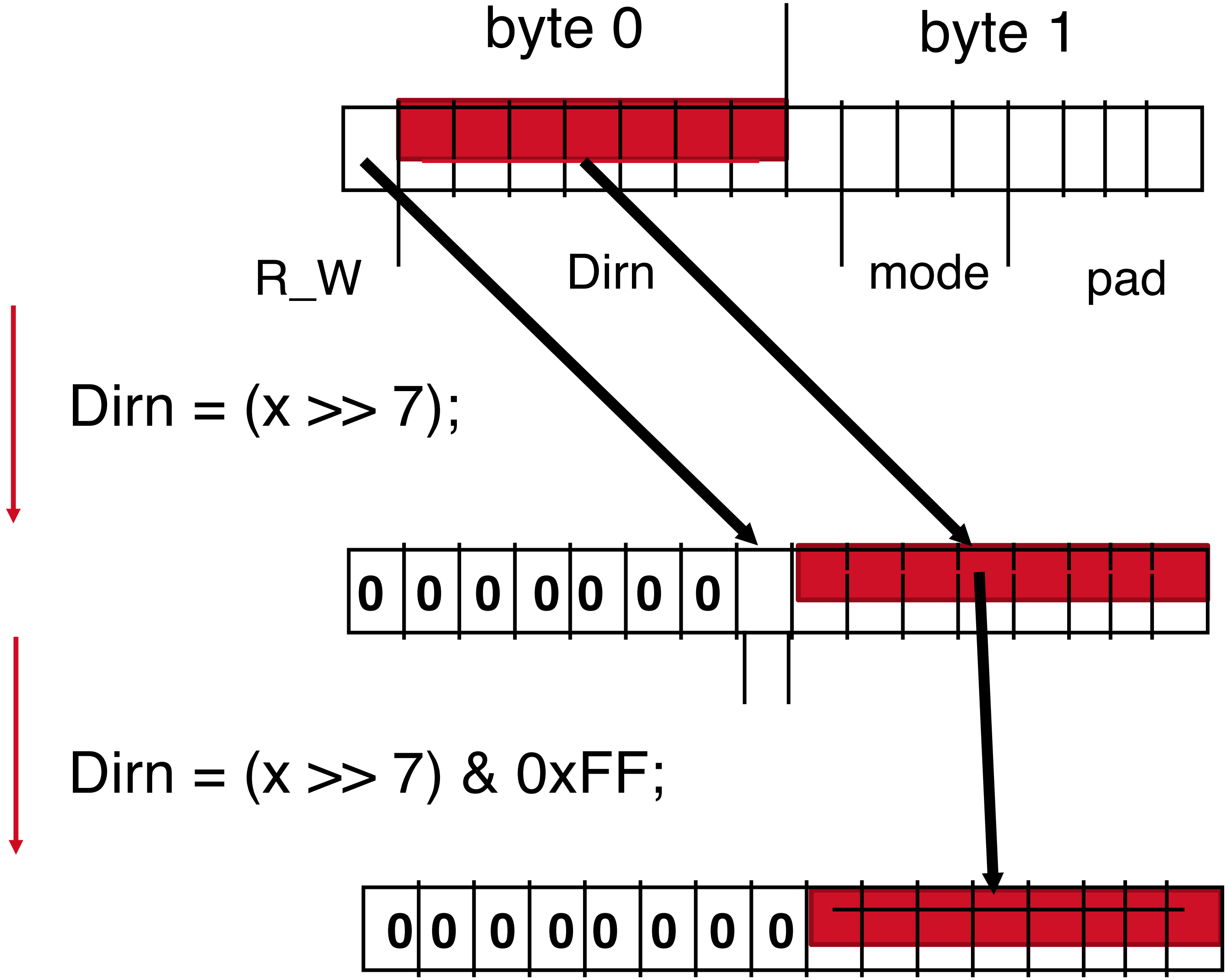
Bit Fields

- 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;
```



byte 0
byte 1
R_W
Dirn
mode
pad
R_W = x >> 15;
byte 0
byte 1
R_W


byte 0
byte 1
R_W
Dirn
mode
pad
Dirn = (x >> 7);
0 0 0 0 0 0 0
Dirn = (x >> 7) & 0xFF;
0 0 0 0 0 0 0 0

Bit Operations

- shift right: >>
- shift left: <<
- bitwise AND: &
- bitwise OR: |
- bitwise XOR: ^
- bitwise NOT: ~
  - Not to be confused with logical NOT !

Beware signed

Summary

› 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

when reading from a file, we asking the operating system to interpret this particular bit in the file, which part of disk I am going to read from

SSH - solid state disk
distributing data in different blacks which maximising its lifetime

- 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)

Operating System help to prevent programs damaging to hardware, acting as a mediator
System Calls are programs to talk to hardware, tends to be expensive as it checks multiple functions before it reach hardware

Files

› Devices are often represented as files

- software reads/write 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

Stream is contiuously area of memory/data (no beginning or end)

- 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*

- data arriving at different rates
- data which cannot be processed until some condition is met, e.g. 25 frame/sec for video streaming

buffering means, save the data until the conditions are satisfied

Files

- For each file opened, there needs to be a file descriptor

Operating System keep record of file descriptor
File descriptor structure is located in c library

- 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 related function
- printf/scanf

Files

type

› **FILE** is a struct that is defined in stdio.h and this is the descriptor

FILE is descriptor

› To open a file, we use the **fopen** function

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

c string

filename

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

variable

mode

myfile will point to file struct somewhere in the memory

path can be relative, or absolute /home/ssta7171/turtles.txt

Files

› `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`

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
e.g. read 10 ints: fread(pinter to memory address, size(int), 10, myfile)

Binary data use

- `fread`
- `fwrite`

› Finish off with `fclose`

Files

- When your program begin, 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**

tell which byte

* Impl. dependent on append

Files

- For reading input files, e.g. **`stdin`**, the end of file is important
  - **`feof()`** tests the end of file indicator

    file end of file = feof
  - EOF does not happen until trying to read beyond end of stream

loop until feof is true

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

byte[0, n-1] —> position of file[0,n]
after reading 5 bytes, the file position is 6th bytes

$ ./printnum < twonum.txt

Files

- 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);
}
```

read the number of tokens sucessfully read from this file and return that numbver

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

Files

- unbuffered – input/output is passed on as soon as possible

  used for devices, require real time feedback, e.g. keyboard
  drawback is very slow, bad performance

- fully buffered – input/output is accumulated into a block then passed

  group elements into ddifferent block and transfer within one block, much more efficient

- 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.

skip input, used for real time application, e.g. games, videoing

Files fgets

- 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…

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

#define BUFLEN (64)

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

```c
int main() {
    FILE *fp = fopen("file.txt", "r");
    if (fp == NULL) {
        fprintf(stderr, "could not open file for reading\n");
        return 1;
    }
    while (!feof(fp)) {
        int num;
        int nread = fscanf(fp, "%d", &num);
        if (nread <= 0) {
            break;
        }
        fprintf(stdout, "num: %d\n", num);
    }
    fclose(fp);
    return 0;
```

```c
int main() {
    FILE *fp = fopen("file.txt", "r");
    if (fp == NULL) {
        fprintf(stderr, "could not open file for reading\n");
        return 1;
    }
    int len;
    char buf[64]; // at most 64 char in a line
    while (fgets(buf, 64, fp) != NULL) {
        len = strlen(buf);
        printf("line is: %s and length is: %d\n", buf, len);
    }
    fclose(fp);
    return 0;
```

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

int main()
{
    struct item {
        char barcode[6]; // 6
        const char *name;  // 8
        float price; // 4
    };

    struct item tomatoes;
    printf("sizeof(struct item): %zu\n", sizeof(struct item)); // 24 bytes
    printf("sizeof(tomatoes): %zu\n", sizeof(tomatoes)); // 24

    struct item *tincan;
    printf("sizeof(struct item *): %zu\n", sizeof(struct item*)); // 8
    printf("sizeof(tincan): %zu\n", sizeof(tincan)); // 8

    tincan = NULL;
    printf("sizeof(tincan): %zu\n", sizeof(tincan)); // 8

    tincan = &tomatoes;
    printf("sizeof(tincan): %zu\n", sizeof(tincan)); // 8

    printf("sizeof(tomatoes.barcode): %zu\n", sizeof(tomatoes.barcode)); // 6
    printf("sizeof(tincan->barcode): %zu\n", sizeof(tincan->barcode)); // 6

    printf("sizeof(tomatoes.name): %zu\n", sizeof(tomatoes.name)); // 8
    printf("sizeof(tincan->name): %zu\n", sizeof(tincan->name)); // 8

    tomatoes.name = "The Greatest Tomatoes in a can";
    printf("sizeof(tomatoes.name): %zu\n", sizeof(tomatoes.name)); // 8
    printf("strlen(tomatoes.name): %zu\n", strlen(tomatoes.name)); // count how many character in the memory // 30

    tomatoes.name = "TGT";
    printf("sizeof(tomatoes.name): %zu\n", sizeof(tomatoes.name)); // 8
    printf("strlen(tomatoes.name): %zu\n",
```

```c
    strlen(tomatoes.name)); // 3

    printf("sizeof(tomatoes.price): %zu\n", sizeof(tomatoes.price)); // 4
    printf("sizeof(tincan->price): %zu\n", sizeof(tincan->price)); // 4

    // pointer arithmetic
    printf("tomatoes: %p\n", &tomatoes); // address of tomatoes
    printf("tomatoes barcode: %zu\n", (void*)&(tomatoes.barcode) - (void*)&tomatoes); // how far off the barcode exist // 0
    printf("tomatoes name: %zu\n", (void*)&(tomatoes.name) - (void*)&tomatoes); // 8
    printf("tomatoes price: %zu\n", (void*)&(tomatoes.price) - (void*)&tomatoes); // 16

    return 0;
}
```

```c
#include <stdio.h>

// idiom
// find the first occurrence of f(x) == TRUE
// where f(x) = (x % 2 == 0)
// return the *both* the value and the index

struct pair {
    int value;
    int index;
};

// if no data then return -1 index;
struct pair get_best_index(int *data, size_t n) {

    struct pair pair;

    pair.index = -1;
    pair.value = -1;

    if (data == NULL || n <= 0) {
        return pair;
    }

    pair.index = 0;
    pair.value = data[0];

    int i;
    for (i = 0; i < n; i++)  {
        int v = data[i];
        if ( v % 2 == 0 ) {
            pair.value = v;
            pair.index = i;
            break;
        }
    }

    return pair; // copy operation
}

int main()
{

    return 0;
}
```

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

struct item {
    char barcode[6];
    const char *name;
    float price;
};

// function prototype
float items_sum( struct item *items, size_t n );

// initialise a structure with values
// pass in the memory address of structure
// Warning: assume name has preallocated memory
void item_init( struct item *item,
    const char *barcode, const char *name, float price) {

    if (item == NULL || // mandatory
        barcode == NULL || // up to programmer
        name == NULL)
        return; // raise an error?

    strncpy(item->barcode, barcode, 6);
    item->name = name; // warning
    item->price = price;
}

int main() {
    // create array
    struct item items[2];

    // initialise elements
    item_init( &(items[0]), "DFH291", "Big tuna", 1.25);
    item_init( &(items[1]), "FGD135", "Tin can", 3.50);

    float sum = items_sum(items, 2);
    printf("sum: %.2f\n", sum); // 4.75

    return 0;
}

// sum all prices
float items_sum( struct item *items, size_t n ) {
    float sum = 0;

    int i = 0;
    for ( ; i < n; ++i) {
        sum += items[i].price;
    }
    return sum;
}

```

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

struct item {
    char barcode[6];
    const char *name;
    float price;
};

// memory input
// given an array of structs

// idiom
// sum all prices
float items_sum( struct item *items, int n )
{
    float sum = 0;

    int i = 0;
    for ( ; i < n; ++i) {
        sum += items[i].price;
        // items[i] == *(items + i + offset of price)
    }
    return sum;
}

int main()
{
    // create array
    struct item items[2];

    // initialise elements
    // man strncpy - see warning
    strncpy(items[0].barcode, "DFH291", 6);
    items[0].name = "Big tuna";
    items[0].price = 1.25;

    //  init each field (man strncpy)
    strncpy(items[1].barcode, "FGD135", 6); // first and second are memory address, third is memory of byte
    items[1].name = "Tin can";
    items[1].price = 3.50;

    float sum = items_sum(items, 2);
    printf("sum: %.2f\n", sum); // 4.75

    return 0;
}
```