More on Pointers, Aggregate types, Files

COMP2129

FACULTY OF
ENGINEERING &
INFORMATION
TECHNOLOGIES

THE UNIVERSITY OF
SYDNEY

Outline

- Revision pointers
- Misc C operators and types
- Aggregate types
  - struct
- Files
  - Finally!

Revision pointers

- What is the value held by p? and how much memory is used by p (in bytes)?
- `int p;`
- `char p;`
- `void foo( int *p )`
- `char *p;`
- `char **p;`

Revision pointers

- What is the value held by p? and how much memory is used by p (in bytes)?
- `int p;`
- `char p;`
- `void foo( int *p )`
- `char *p;`
- `char **p;`
- `int **p;`
- `long *p;`
- `void *p;`
- `const unsigned long long int * const p;`
- `bubblebobble ************p;`

Pointer interpretation

- `char *p`
  - Address to a single char value
  - Address to a single char value that is the first in an array
- `char *argv[]`
  - Array of "the type" with unknown length
  - Type is `char *`

- `char **argv`
  - `*` Address to the first element to an array of type char *
  - Then, each element in `*` is an…
    - `*` address to the first element to an array of type char

char argv[][3]; // oh no!

Pointer interpretation

› Interpretations of `int **data;`

1. Pointer to pointer to single int value
2. Array of addresses that point to a single int
3. Address that points to one array of int values
4. Array of addresses that point to arrays of int values

Pointer interpretation

- Interpretations of `int **data;`
  1. Pointer to pointer to single int value
  2. Array of addresses that point to a single int
  3. Address that points to one array of int values
  4. Array of addresses that point to arrays of int values

- Thinking about each `*` as an array:
  1. Array size ==1, Array size ==1
  2. Array size >=1, Array size == 1
  3. Array size ==1, Array size >= 1
  4. Array size >=1, Array size >= 1

Pointer comments

- When you call a function in Java, compare passing a primitive type and Object type.

- You may have heard:
  - Pass by value
  - Pass by reference

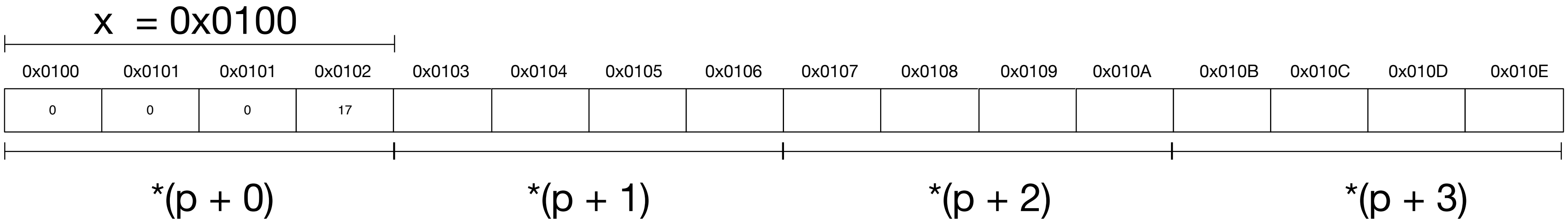
What is the meaning of this in C?

- void has no size, but sizeof(void*) is the size of an address

- Pointers are unsigned numbers, why?

Pointer arithmetic

- int *p = NULL;
- int x[4];
- p = x;



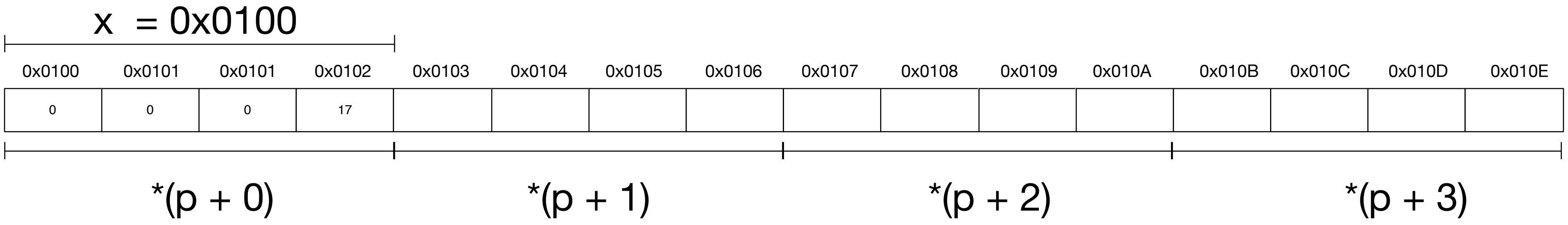
x = 0x0100

0x0100	0x0101	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106	0x0107	0x0108	0x0109	0x010A	0x010B	0x010C	0x010D	0x010E
0	0	0	17												

- Seeking to the nth byte from a starting address?

Pointer arithmetic

- int *p = NULL;
- int x[4];
- p = x;



x = 0x0100

0x0100	0x0101	0x0101	0x0102	0x0103	0x0104	0x0105	0x0106	0x0107	0x0108	0x0109	0x010A	0x010B	0x010C	0x010D	0x010E
0	0	0	17												

*(p + 0) *(p + 1) *(p + 2) *(p + 3)

- Seeking to the nth byte from a starting address?

```
void *get_address( sometype *data , int n) {
     unsigned char *ptr = (unsigned char*)data;
     return (void*)(ptr + n);
}
```

sizeof operator

- Not all h/w architectures are the same
  - different sizes for basic types

- C specification does not dictate exactly how many bytes an int will be

- **sizeof** operator returns the number of bytes used to represent the given type or expression
  - `sizeof( char )`
  - `sizeof( int )`
  - `sizeof( float * )`
  - `sizeof ( 1 )`
  - `sizeof( p )`

sizeof operator

› Not all h/w architectures are the same

- different sizes for basic types

› C specification does not dictate exactly how many bytes an int will be

› **sizeof** operator returns the number of bytes used to represent the given type or expression.

- `sizeof( char )`
- `sizeof( int ), sizeof( double )`
- `sizeof( float * )`
- `sizeof ( 1 ), sizeof ( 1/2 ), sizeof (1.0 / 2.0)`
- `sizeof( p ) ????`

sizeof

- Special case for **p**, what is it?
  - `char p;`
  - `char *p;`
  - `char p[8];`

- But…
  - `char msg[100];`
  - `char *p = msg;`
  - `char msg2[] = “got my goat and up my nose”;`
  - `char *p = msg2;`
  - `char *p = “im finna spaz on anybody now”`

- **sizeof** needs to be used carefully

Less familiar types

- The types **char** will support the value range from CHAR_MIN to CHAR_MAX as defined in file <limits.h>
  - `#define UCHAR_MAX      255        /* max value for an unsigned char */`
  - `#define CHAR_MAX       127           /* max value for a char */`
  - `#define CHAR_MIN       (-128)        /* min value for a char */`

- Most C implementations default types as `signed` values, but a warning that you should not assume this.
- **unsigned** and **signed** enforce the sign usage
  - `char ch;`
  - `signed char ch;`
  - `unsigned char ch;`
  - `unsigned int total;`

Less familiar `const`

- `const` prevents the value being modified
  - `const char *fileheader = “P1”`
  - `fileheader[1] = '3';` Illegal: change of char value

- It can be used to *help* avoid arbitrary changes to memory

- The value `const` protects depends where it appears
  - `char * const fileheader = “P1”`
  - `fileheader = “P3”;` Illegal: change of address value

- Reading right to left:
  - Is an address, points to a char, that is constant
  - Is an address, that is constant

Less familiar `const`

- `const` prevents the value being modified
  - `const char *fileheader = “P1”`
  - `fileheader[1] = '3';` Illegal: change of char value

- It can be used to *help* avoid arbitrary changes to memory

- The value `const` protects depends where it appears
  - `char * const fileheader = “P1”`
  - `fileheader = “P3”;` Illegal: change of address value

- You can cast if you know if the memory is writable

writable

```
char fileheader[] = {'P', '1'};
const char *dataptr = (char*)fileheader;
char *p = (char*)dataptr;
p[1] = '3';
```

Non-writable

Floating point types

- Exact bit representation unknown, usually IEEE 754
- Generally, floating point number $\mathbf{x}$ is defined as:

$$x = sb^{e} \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \leq e \leq e_{\max}$$

- s sign
- b base of exponent (e.g. 2, 10, 16)
- e exponent
- p precision
- f_k nonnegative integer less than b

+0	-0
+ve / 0 = +infinite	-ve / 0 = -infinite
NaN (not a number)	Zero exponents…

Enums

COMP2129

FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

THE UNIVERSITY OF SYDNEY

The picture so far – simple types

- simple data types:
  - int, char, float.....

- pointers to simple data types:
  - int *, char *, float *

Enumerated types

- enums (enumerated types) are another simple type
- enums map to int
- an enum associates a name with a value

Enumerated types

```
enum day_name
{
  Sun, Mon, Tue, Wed, Thu, Fri, Sat, day_undef
};
```

- Maps to integers, 0 .. 7
- Can do things like ‘Sun ++'
- very close to int

Enumerated types

```
enum month_name
{
   Jan, Feb, Mar, Apr, May, Jun,
   Jul, Aug, Sep, Oct, Nov, Dec,
   month_undef
};
```

Enumerated types

- we could always use integers to represent a set of elements

- but enums make your code much more readable

- eg red instead of 0

  - How many bytes for an array of enum?

Structures

COMP2129

FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

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

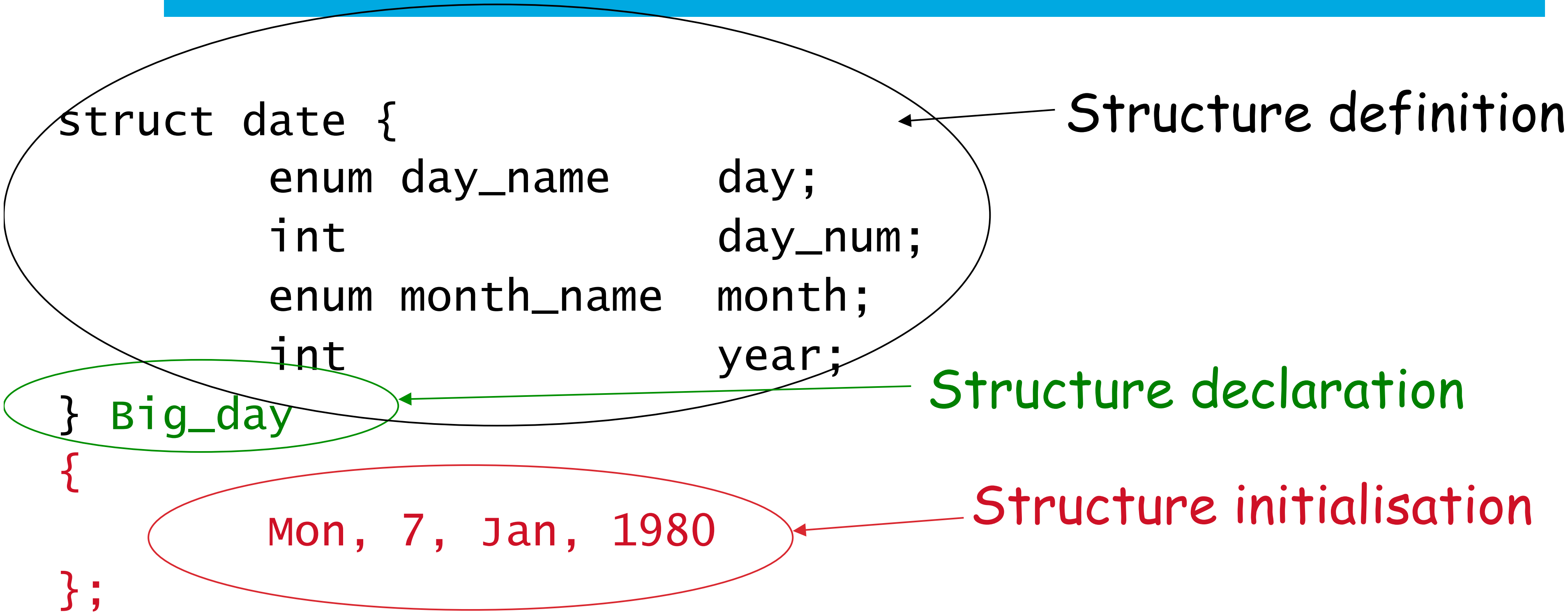
name of the type of structure

```
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
};
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
};
```

Structure definition

Structure declaration

Structure initialisation

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

Structures

```
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;

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

```
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;
}
```

Standard structures

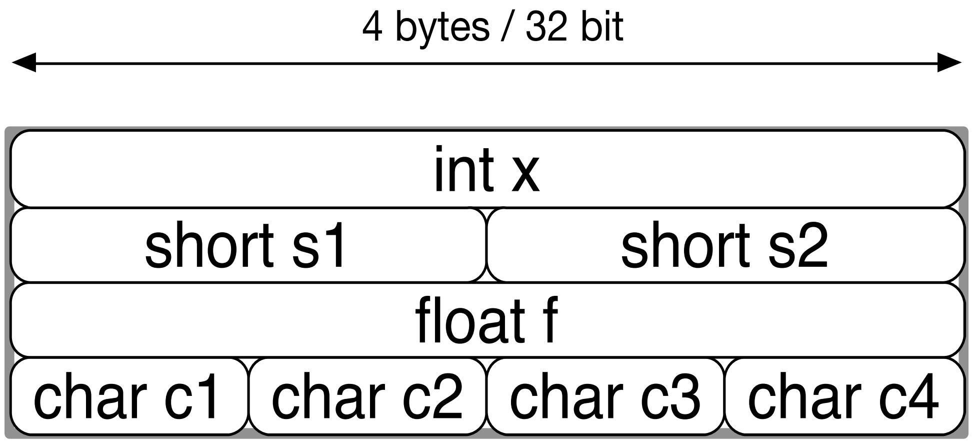
- `stdio.h`
- `time.h`
- `stat.h`
- `pwd.h`

```
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 */
```

```
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;
};
```



4 bytes / 32 bit

int x

short s1 short s2

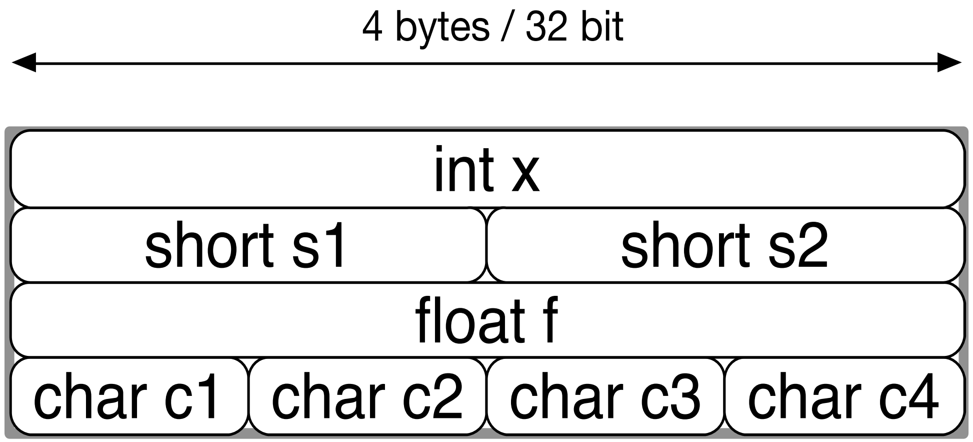
float f

char c1 char c2 char c3 char c4

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

Memory alignment

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



4 bytes / 32 bit

int x

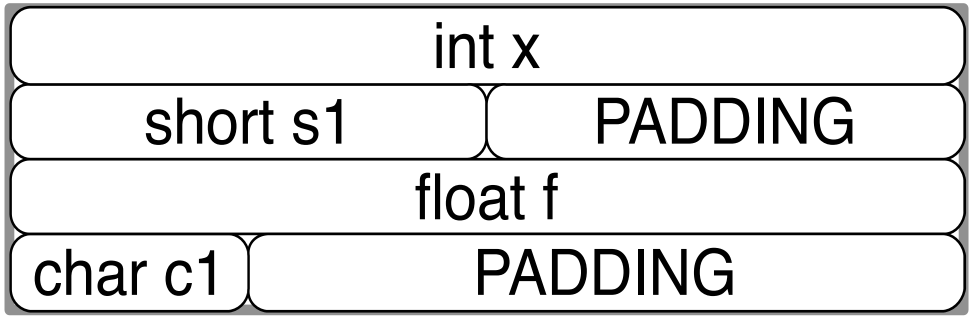
short s1 | short s2

float f

char c1 | char c2 | char c3 | char c4

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

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



int x

short s1 | PADDING

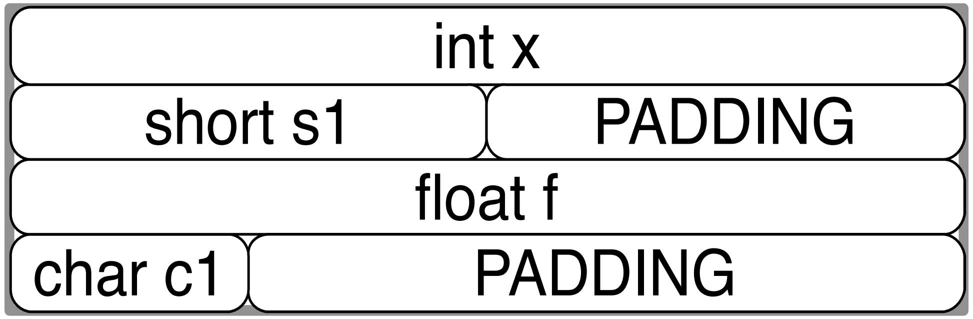
float f

char c1 | PADDING

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

Memory alignment

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



int x

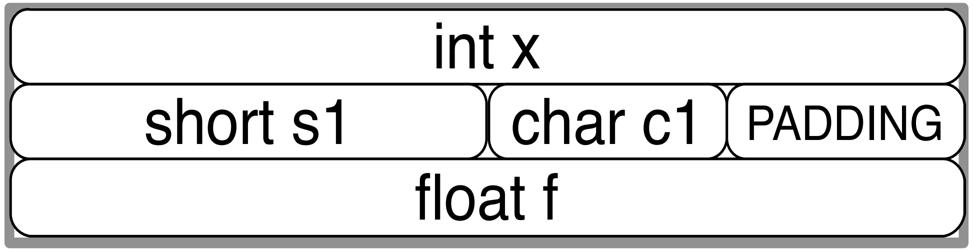
short s1 | PADDING

float f

char c1 | PADDING

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

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



int x

short s1 | char c1 | PADDING

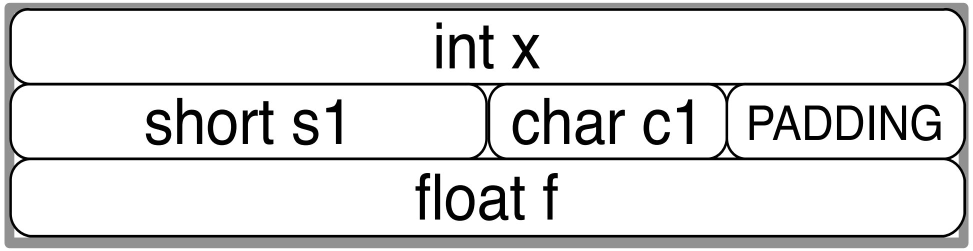
float f

```
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

COMP2129

FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

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

```
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

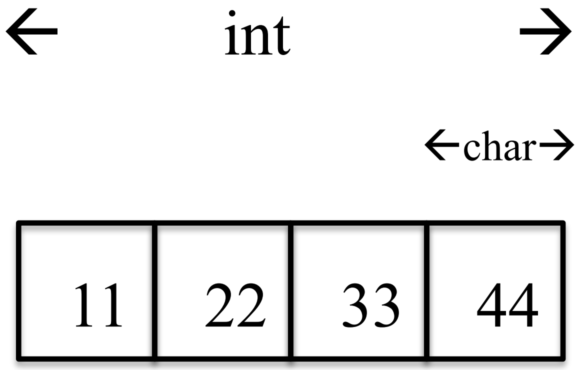
```
enum holding_type {book, film};
struct catalog
{
        char *  title;
        enum holding_type type;
        union
        {
                struct /* book */
                {
                        char * author;
                        char *  ISBN;
                } book_info;

                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:



← int →

←char→

11 22 33 44

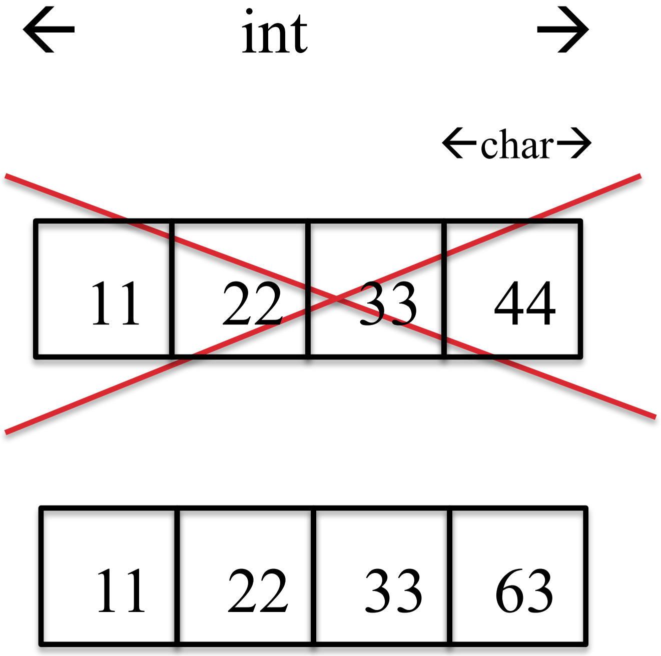
```
union
{
        int     a;
        char    b;
} x;

x.a = 0x11223344;
```

THE UNIVERSITY OF SYDNEY

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;
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;
}
```

Files in C

COMP2129

FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

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
  - Files have 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)

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
  - 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*

Files

- 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

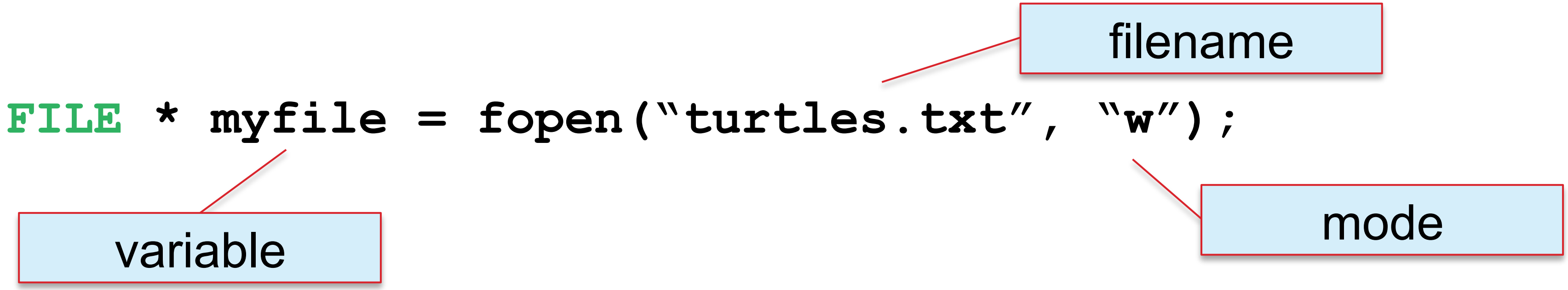
```
FILE *fopen(const char *path, const char *mode);
```

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

Files

- 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");
```

filename

variable

mode

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`**
- Binary data
  - **`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`**

* 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
  - 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);
}
```

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);
}
```

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

Files

- 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 streams: force write all data,
  - Input streams: discard any unprocessed buffered data.

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…

fgets example

```
#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;
}
```