CSSE2310/7231 — 3.1

Multi-dim arrays, FILEs,
Recursion,
Values with names. . .

Multi-dim arrays

Eg: An $M * N$ array of `int`.
Three options:

1. `int arr[M][N]` — 2D array, size fixed at compile time
   - Not convertable to `int*`
2. `int* arr = malloc(sizeof(int)*M*N)` — fake it with a 1D array
3. `int** arr = malloc(sizeof(int*)*M)` — array of arrays

Fake with 1D

```
int* arr = malloc(sizeof(int)*M*N);

// lookup arr[i][j]
arr[i*M+j]

free(arr);
```

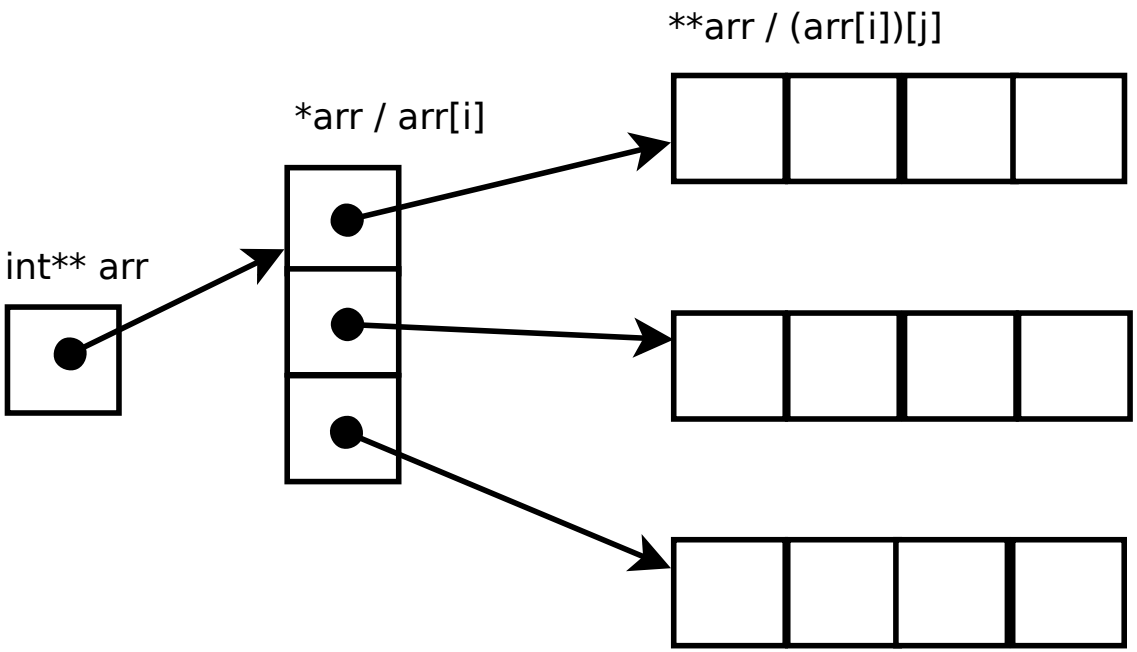
Array of arrays

```
int** arr = malloc(sizeof(int*)*M);
for (int i=0;i<M) {
    arr[i]=malloc(sizeof(int)*N);
}

// lookup arr[i][j]
arr[i][j]

for (int i=0;i<M;++i) {
    free(arr[i]);
}
free(arr);
```

Array of arrays



FILEs

FILE*

- The type for C standard I/O is `FILE*`.
  It should be treated as an *opaque* type[1]
- To interact with a file, use `fopen()` to get a `FILE*` for it.
- `fclose()` the FILE* when you are finished with it.

[1] ie don't try to dereference it or look inside.

Example, copying files

One byte at a time:

See `copyf.c`

- `fgetc()` takes a file and reads one char from it.
  - It returns either a `char` or the special value `EOF`
  - This is why it needs to be `int` (to store any possible char AND an extra value).

When is it eof?

`feof(f)` is true when the program has tried to read from `f` and failed because it was at the end.

C tests for the edge of a cliff by asking "are we falling?".

comma operator for fun and profit

See `copyf2.c`

Evaluating: *expr1*`,` *expr2*

- evaulate *expr1*
- throw the result away
- evaluate *expr2*

Is `(5+3), 7` pointless?

- Yes.
- Useful if the first expression has *side effect*[2]
- eg: `++things, things > 2`

[2] The expression does something other than just making a value

Error checking

See `copyf3.c`.

Error checking

`fopen()` returns the null pointer if it can't open the file.
It will set the `errno` variable to indicate what the problem was.

```
#include <errno.h>

    FILE* fin = fopen(...);
    if (fin == 0) {
        perror("Opening file:"); // what happened
        return ...
    }
```

It will give a name you can look up in man pages

output ffunctions

- `fprintf(FILE*, const char* format, ...)`
- `fputc()`
- `fputs()`
- `fwrite()`

Consult man pages for parameter order.

input ffunctions

- `fgetc()`
- `fgets()`
- `fread()`

`fscanf()`

See `scandemo.c`

Notes:

- Need to pass pointers to the variables you want to input into.
- Need to use a different placeholder for doubles.

Yes there is a `scanf()`.

`fscanf()`

Not as wonderful as it looks.
In particular it makes error handling difficult.

Consider using `sscanf()` instead.

See `scandemo2.c`

buffered output

See `buffo.c`.

Just you have printed something doesn't mean it has actually left the buffer yet.

See `buffo2.c`

"Hey I found a way to disable buffering..."
Generally not a good idea. Buffering exists for a reason.

What if I don't close?

- There is a system limit on how many files you can have open at one time. A long-running program with lots of files open might prevent you from opening any more.
- If your program exits "normally" ie:
  - return from `main()` or
  - call `exit()`

  All open `FILE*`s will be closed[3].
- If your program terminates abnormally, then the file will be closed but no flushing will occur.

[3] and flushed for output files

Recursion

Recursion

A recursive defintion is one which defines something in terms of itself.

Common examples starting examples are:

- Factorial : $n! == 1 * 2 * 3 * \ldots (n-1) * n$
  - $f(n) == f(n-1) * n$
- Fibonacci : $f(n) == f(n-1) + f(n-2)$

Factorial

```
unsigned long fac(unsigned long n) {
    if (n<=1) {              // ``base'' / ``stopping'' case
        return 1;
    }
    return fac(n-1)*n;   // recursive case
}
```

Fibonacci

```
unsigned long fib(unsigned long n) {
    if (n==0) {
        return 1;
    } else if (n==1) {  // can have multiple base cases
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Recursion part1

A recursive function should have:

- One or more base cases
- At least one recursive call
  - Must be to a smaller / "easier" instance of the problem.
  - Must hit a base case eventually.

Recursive vs iterative

Recursion pitfalls:

- Repeated work (eg Fibonacci)
- Blowing stack
  - A 1,000,000 iteration `for` loop doesn't take more memory than a 10 iteration loop[4].
  - A 1,000,000 deep recusive calls is a problem.

Claim: any recursive algorithm can be converted to an iterative algorithm.
So why use recursion?

- In some cases it is just clearer

[4] probably

Merge sort — concept only

Sort an array:

1. Divide array into two halves.
2. Sort each half
3. Merge the halves
   - Look at the first element in each sorted half
   - Move the a minimal one into a new array
   - Repeat until all elements moved

If only we knew how to sort, half an array.

Merge sort

Base cases:

- An empty array is already sorted
- An single element array is already sorted.

So:

Let A be an array, n is the number of elements in A, W is an array which is at least as big as A

```
void S(A, n, W) {
    if (n<=1) {
        return;
    }
    S(A,n/2, W);              // sort first half
    S(A+(n/2), n-n/2, W);// sort second half
    Merge(W, n, A);
}
```

Why W?

Sort needs another array to copy the sorted output into prior to merging it back.
We could do this:

```
void S(A, n, W) {
    if (n<=1) {
        return;
    }
    W=malloc(n*...)
    S(A,n/2, W);              // sort first half
    S(A+(n/2), n-n/2, W);// sort second half
    Merge(W, n, A);
    free(W);
}
```

But mallocing every time the function is called seems overkill. So instead we require the caller to supply us with an array we can use.

Wrapper functions

Our recursive call requires an extra array, but we don't want programmers calling our function to have to worry about extra setup. We want to be able to call something like `S(A,n)`.
Rename `S` $\rightarrow$ `SRec`

```
void MSort(A, n) {
    W=malloc(n*sizeof(...));
    SRec(A,n,W);
    free(W);
}
```

Lesson = It's ok to write a setup routine that calls your real code but deals with setup and extra params first.

Values with names

Booleans

Boolean logic is a logic where there are only two possible values.

- {`1`,`0`}
- {High, Low}
- {`true`, `false`}[5]

To use the **literals** `true`, `false` and the type `bool` in your programs, you need to `#include <stdbool.h>`.

You don't need this header just to use expressions which have true/false results.
eg: `if (x==4)`

Why not?

[5] Python programmers note: lower case

Boolean "adjacent" things

In C any expression which produces a numeric value[6] can be interpretted as "true" or "false".

$==0 \rightarrow$ `false`

$!=0 \rightarrow$ `true`

[6] Most things in C boil down to numbers (eg pointers)

Boolean "adjacent" things

This leads to some weird results:

```
int i=0;
if (++i) {
    // this will execute
}
i=0;
if (i=0) {
  // this will not execute (note the = not == )
}
```

An assignment expression evaluates to the value of the right hand side.

```
if (x=7) // true
if (fgets(buffer, size, inputfile))
    // true if input read, false if end of file
```

Can use this for test and capture.

eg: suppose f() returns 0 on error.

```
if (x=f()) {
    // use x
} else {
    // failure case
}
```

Preprocessor macros

The preprocessor runs before the main compile and deals with # directives.

```
#define PI 3.141
```

Every occurance of `PI` will be replaced with 3.141. *The compiler will never actually see `PI` it will only see the replacement value*.

This can create problems using a debugger because the code you see is not exactly the code that was compiled.

Macros with parameters

```
#define CUBE(X) ((X)*(X)*(X))
```

- These can look like a function call, but are expanded by the preprocessor.
- Be careful with ()
  - Could we have `#define CUBE(X) X*X*X` ?
  - Yes. But...
  - `CUBE(2+3)` $\rightarrow$ `2+3*2+3*2+3` $==$ 17, not 125
- Beware of side effects:
  - `int x=1; int y=CUBE(++x);`
  - If it were a function, we would expect the answer to be 8.
  - `(++x)*(++x)*(++x) = 2*3*4 = 24` (and you've broken x)

Why???

Beyond the scope of this course.

- More useful in embedded settings(?)
- Can mark functions as inline to hint to the compiler that it should do this on proper functions.

Conditional compile

```
#define BOB
```

Tells the preprocessor that "BOB" is a symbol it should recognise but doesn't actually give it a value. These can also be defined on the commandline with `-DBOB`.
Conditional compile:

```
#ifdef OMP_SUPPORT
void stuff_that_only_works_under_omp();
#endif
```

Include guards:

```
#ifndef BOB_H
#define BOB_H

// Bob things
#endif
```

See `h1.h`, `h2.h`, `h.c` vs `g1.h`, `g2.h`, `g.c`

enums

`bool` allows us to have a variable which stores one of a set of named values {`true`, `false`}.
What about:

- days of the week? {`SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`}
- states in a statemachine? {`SETUP`, `CONNECTED`, `WORKING`, `DISCONNECTED`, `ERROR`}
- ...

enums

```
enum Day {
    SUNDAY,
    MONDAY,
...
};

enum Day d=TUESDAY;
```

Behind the scenes, the compiler will choose an `int` value for each member of the enum.

Fixed values?

```
enum Errors {
    E_OK = 0,
    E_TOO_MUCH = 1,
    E_NOT_A_NUMBER = 2
};
```

switch

```
enum State s;

switch (s) {
case SETUP:
    printf("Doing setup\n");
    break;
case CONNECTED:
    printf("Doing connected\n");
    printf(" more connected\n");
    break;
default:
    printf("Doing other states\n");
};
```

If a case does not end in a `break`, the program will execute the next case as well.

switch

See `switch.c`

Switch can be used with any integer type.
So it doesn't work with strings or floats.

`break` and `continue`

break

`break` will jump out of the inner-most, loop or `switch` statement.

```
for (int num=2; num<100; ++num) {
    bool prime=true;
    for (int factor=2; factor<num; ++num) {
        if (num%factor == 0) {
            prime=false;
            break;
        }
    }
    if (prime) {
        printf("%d ", num);
    }
}
```

continue

`continue` jumps to the next iteration of the inner-most loop.

```
while (fgets(buffer, 80, input)) {
    if (strlen(buffer)<5) {
        continue;   // read the next line
    }
    // more processing
}
```

Operators and types

Operators

	Equivalent to	**Evaluates to**
`a += b`	$a_{\text{new}} = a_{\text{old}} + b$	$a_{\text{old}} + b$
`a *= b`	$a_{\text{new}} = a_{\text{old}} * b$	$a_{\text{old}} * b$
...		
`a << b`	not in this course	
...		
`++i`	$i_{\text{new}} = i_{\text{old}} + 1$	i_{new}
`i++`	$i_{\text{new}} = i_{\text{old}} + 1$	i_{old}
`--i`	$i_{\text{new}} = i_{\text{old}} - 1$	i_{new}
`i--`	$i_{\text{new}} = i_{\text{old}} - 1$	i_{old}

Bitwise Operators

- a|b (or) — 6|10 == 14
- a&b (and) — 6&10 == 2
- a^b (xor) — 6^10 == 12

Logical Operators

	value	**calls**
f1() || f2()	false	f1, f2
t1() || t2()	true	t1
f1() || t1()	true	f1, t1
f1() && t1()	false	f1
t1() && f2()	false	t1,f1
t1() && t2()	true	t1,t2

Types

- Signed integer types:
  char[7] $\leq$ short int $\leq$ int $\leq$ long int $\leq$ long long int
- unsigned integer types[8]:
  unsigned char $\leq$ unsigned short int $\leq$ unsigned int $\leq$
  unsigned long int $\leq$ unsigned long long int
- floating point types
  float $\leq$ double $\leq$ long double ...
- boolean:
  bool, ... numeric types

[7] maybe

[8] the "int" is optional

Function pointers

Why?

Sometimes we want to put functions into variables:

- Callbacks : when a particular thing happens call this function
  - GUIs and other event driven tasks
- Flexible functionality
  - Change how one part of an overall task is done.
  - Eg: sorting how is ordering defined? (see `man 3 qsort`)
- ...

In C the name of a function (without () is treated as a function-pointer for it).

- Similarly to the name of an array being a pointer to the first element.
- See `fp1.c` for a possible error message.

syntax

The type of `g` is:
`int (*)(void)`

A function which takes two `int`s and returns an `int` has type:
`int (*)(int, int)`

`(*)(...)` is your cue that a function pointer is involved.

syntax

The name of the variable[9] goes with the `(*)`

`int (*vname)(int, char)`
`vname` is a variable storing a pointer to a function which returns an int and takes an `int` and a `char` as params.

See `fp2.c`.

See `fp3.c`.

[9] or name of type for `typedefs`

syntax

See why people might want to use `typedefs`: `fp4.c`.

What about:

```
void qsort(void* base, size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));
```

Examples

Examples

```
void* (*(*var)(int, int))(char*)
```

- var is a pointer to a function which takes two `int`s and returns a pointer to function taking a `char*` and returning a `void*`.
- ```
  typedef void* (*ft)(char*);
  ft (*var)(int, int)
  ```
- In this course you are permitted to use typedefs to simplify if needed.
- `http://cdecl.org` for practice.
  ```

Unions

`unions` — not in 2310

Problem: want to be able to store different types of things in a variable. eg: Suppose a person at UQ could be:

- A staff member
- A coursework student
- A research student
- A study abroad student

Different types of people might need different information stored about them.

unions — staff

```
struct UQStaff {
    short pType;
    char* ID;
    char* name;
    char* unit;
    char* supervisorID;
    float fteFraction;
};
```

unions — coursework

```
struct UQCWStudent {
    short pType;
    char* SID;
    char* name;
    char* program;
    CourseList* enrolment;
    time_t predictedGraduation;
};
```

unions — coursework

```
struct UQRStudent {
    short pType;
    char* SID;
    char* name;
    char* supervisorID;
    char* program;
};
```

unions — abroad

```
struct UQAbroadStudent {
    short pType;
    char* SID;
    char* name;
    CourseList* enrolment;
    char* homeInstitution;
};
```

unions

We want to be able store any of these in a variable.
`void*`?
That would work, but a void pointer could be anything.
We want a type that can only hold one of the following structs:

- UQStaff
- UQCWStudent
- UQRStudent
- UQAbroadStudent

In an OO language you could make these subclasses.

unions

```
union UQPerson {
    struct UQStaff staff;
    struct UQCWStudent cw;
    struct UQRStudent res;
    struct UQAbroadStudent ex;
};
```

Memory layout

See `un.c`

Unlike the members of a struct, members of a union all occupy the same memory.

There will not be an exam question on unions[10].

[10] In this course