CSSE2310/7231 – Lecture Week 3

C continued
Multi-dim arrays, FILEs, ...

Peter Sutton
School of Electrical Engineering and Computer Science
The University of Queensland

Outline

- Admin
- Multi-dimensional arrays
- Interacting with files
- Preprocessor
- `enum`, `switch`, `break`, `continue`
- types, function pointers

Admin

- Pracs from now on will use queueing system - go to https://q.uqcloud.net

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

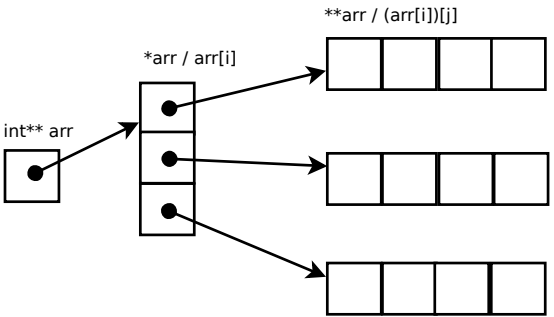
Fake with 1D

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

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

free(arr);
```

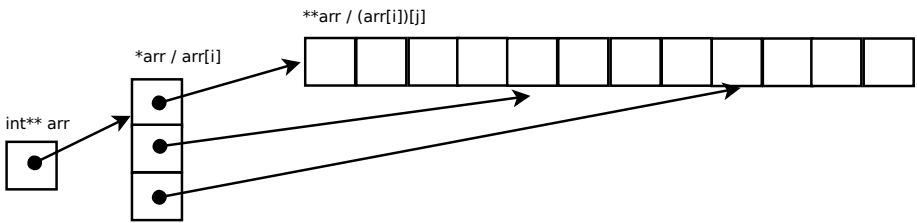
Array of arrays

```
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



Array of arrays - another option!



Details can matter!

What's the difference?

```
for (int r=0; r<SIZE; r++) {
    for (int c=0; c<SIZE; c++) {
        arr[r][c] = 0;
    }
}
```

vs

```
for (int c=0; c<SIZE; c++) {
    for (int r=0; r<SIZE; r++) {
        arr[r][c] = 0;
    }
}
```

See `stride.c` (and make friends with the `time` utility)

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.
- `stdio.h` defines three special `FILE*` variables that are always available
  - `stdin`
  - `stdout`
  - `stderr`

[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).
  - For fun, try `printf("%d\n", EOF)`

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.

`stdio` 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`.

- Note also – you don't need to use the comma operator ...

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.

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

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

See `copyf4.c`

output functions

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

Consult man pages for parameter order.

input functions

- `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()` (cont.)

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`.

Aside: '`watch`' - a useful shell command

`$ watch -n` *`seconds command`*

Just because 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

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)` → 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 use macros with parameters?

Beyond the scope of this course.

- Used carefully macros can reduce repetitious and complex code structures, but can be very hard to debug
- Often used to force inline function code (faster) but modern compilers recognise the `inline` keyword

Preprocessor abuse is a staple technique in the *International Obfuscated C Competition* - `https://www.ioccc.org/`

See `ditdah.c`

`donut.c` and `carlini.c` are fun too

Conditional compilation

```
#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 `gcc` commandline with `-DBOB`.
Conditional compile:

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

Most frequent usage - Include/header 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`

Things to watch for

- Switch can be used with any integer-like type - doesn't work with strings or floats.
- `case` statements must be constants.
- Missing `break` statements!

If you are feeling brave, check out *Duff's Device* but please don't use it in your assignments :)

`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
}
```

Types

Types

- Signed integer types:
  char[4] $\leq$ short int $\leq$ int $\leq$ long int $\leq$ long long int — see `size.c`
- unsigned integer types[5]:
  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

The header file `<limits.h>` defines the min and max values for most (all?) types, regardless of architecture

[4] maybe

[5] the "int" is optional

<limits.h> excerpt

```
/* Number of bits in a `char'.  */
#  define CHAR_BIT      8

/* Minimum and maximum values a `signed char' can hold.  */
#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127

/* Maximum value an `unsigned char' can hold.  (Minimum is 0.)  */
#  define UCHAR_MAX     255

/* Minimum and maximum values a `signed short int' can hold.  */
#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

...
```

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 warning message (with `-Wall`)

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[6] 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`.

[6] or name of type for `typedef`s

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

What type is `var` in each of the following:

- `char* var[]`
- `long var[10]`
- `int** var[10]`
- `void (*var)(int, double)`
- `int *(*var[5])()`
- `void (*var)(int* (*)(int), int)`

Examples

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

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*`.

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)
  ```
  ```

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.
  ```

Coming up

- Friday Contact – More C (cont.) ...
- You should be working on ...
  - C exercises
  - Subversion and make exercise
  - Assignment 1