About the test

[Learn Posting]

[Sample test]

10. C potpourri

- Some other C goodies that were skimmed over or skipped.
  - File I/O (skimmed over earlier)
  - Declaration qualifiers
    - *static, extern, volatile, register*
  - Conditional compilation
  - Multidimensional arrays
  - Passing functions as parameters
  - Bit manipulation
  - Other standard libraries

I/O Streams

- An I/O stream abstracts the idea of a sequential file.
- Delivers (input stream) or accepts (output stream) a stream of bytes.
- C type *FILE** defined in *stdio.h*
- Open a file with *fopen*

```
    FILE* fIn = fopen("sillyIn.txt", "r");
```

File name

"r" = open for reading
"w" = open for writing
"a" = open for append
+ others ; see docs

main I/O functions

- Character-by-character

```
int fgetc(FILE* fp)
int fputc(int c, FILE* fp)
getchar() = fgetc(stdin)
putchar(c) = fputc(c, stdout)
```

Return an *int* with char in low byte, or the special value EOF (-1)

- Line-by-line

```
char* fgets(char* buffPtr, int size, FILE* fp)
int fputs(const char* buffPtr, FILE* fp)
puts(s) = fputs(s, stdout)
```

  – There's also a *gets* but you should *NEVER* use it!

- Warning: line-termination is platform-dependent.
  – Linux, MacOS X: \n
  – Windows: \r\n
  – Old Mac: \r

Creates problems when moving text files between OSs

main I/O functions (cont'd)

- Block-by-block (binary files)

```
size_t fread(void* buffPtr, int elemSize, int numElems, FILE* fp)
size_t fwrite(const void* buffPtr, int elemSize, int numElems, FILE* fp)
```

- Formatted I/O

```
int fscanf(FILE* fp, const char* format, ...)
int fprintf(FILE* fp, const char* format, ...)
printf(s, ...) == fprintf(stdout, s, ...)
scanf(s, ...) == fscanf(stdin, s, ...)
sprintf(char* buffPtr, const char* format, ...)
snprintf(char* buffPtr, int buffsize, const char* format, ...)
```

sprintf is dangerous (buffer overflow)
but *snprintf* is not ANSI :-(

Declaration qualifiers

- *static*
  1. Applied to a global-level variable: means “visible only within this file”
     - So isn’t passed to the linker within the .o file
  2. Applied to a local variable: means isn’t an *auto* variable (i.e., within the stack frame) but occupies space in the initialised data segment of the program
     - So holds its value over multiple calls to function
- *extern*
  - Tells the linker that the *definition* of a variable is elsewhere (i.e., it’s not allocated space within this module, but can be referenced).

Declaration qualifiers (cont'd)

- *volatile*
  - Tells the compiler that this variable's value may change unpredictably so don't use the optimiser on it
    - Presumably because of some hardware or software hack that gives an external agent access to the variable
- *register*
  - Suggests to the compiler that this value could usefully be held in a register (for maximum efficiency)

Conditional compilation

- “if” statements in preprocessor to control what gets passed to the compiler, e.g.:

```
#ifdef DEBUG
printf("Allocated a new student, name = %s\n", stud.name);
#endif
```

- Can define the symbol DEBUG at the top with

```
#define DEBUG
```

or at compilation time with the –D flag to gcc, e.g.

```
gcc $CFLAGS -DDEBUG sillyprog.c -o sillyprog
```

- Widely used through C library code, e.g., to enable special GNU language extensions

2D arrays revisited

Two sorts of 2D arrays:

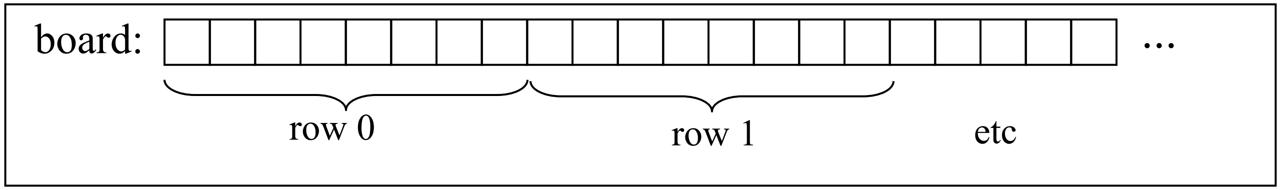
(a) 1D arrays with fudged subscripting

```
typedef enum{PAWN, BISHOP, KNIGHT, ROOK, KING, QUEEN} Piece;

void setPiece(int board[8][8], int row, int col, Piece piece)
{
    board[row][col] = piece;
}

int main(void) {
    int board[8][8];
    setPiece(board, 3, 5, ROOK);
}
```

Compiler needs at least second subscript to compute actual 1D subscript as *row*numCols* + *col*



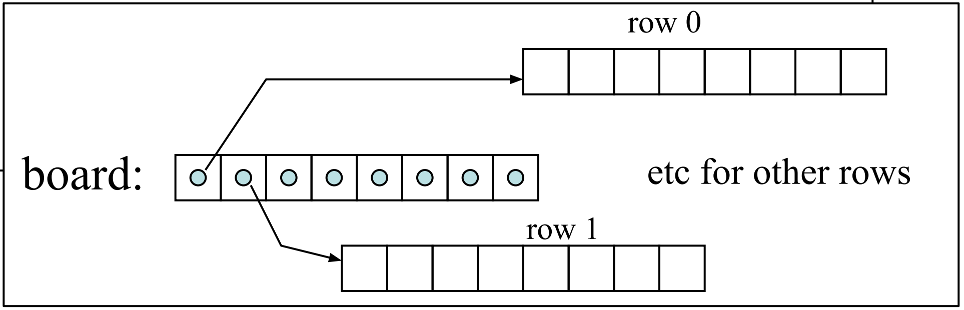
2D arrays (cont'd)

(b) *vectored* arrays (Java style)

```
typedef enum{PAWN, BISHOP, KNIGHT, ROOK, KING, QUEEN} Piece;

void setPiece(int** board, int row, int col, Piece piece)
{
    board[row][col] = piece;
}

int main(void)
{
    int* board[8];  // an array of 8 pointers to ints
    for (int row = 0; row < 8; row += 1) {
            board[row] = calloc(8, 4);  // Allocate space for a
row
    }
    setPiece(board, 3, 5, ROOK);
    return EXIT_SUCCESS;
}
```



Function pointers

Useful for “callbacks” or abstracting behavioural details etc, e.g. see *qsort* function

```
void sillyFunc1(const char* name)
{
    printf("Hello %s\n", name);
}

void sillyFunc2(const char* name)
{
    printf("Goodbye %s\n", name);
}

void doSomethingElse(void (*f)(const char*), char* arg)
{
    f(arg);
}

int main(void)
{
    doSomethingElse(&sillyFunc1, "Richard");
    doSomethingElse(&sillyFunc2, "Fred");
}
```

Bit manipulation

- Operators '`|`', '`&`' and '`^`' are bitwise OR, AND and XOR, resp.
- Often used to encode booleans as bits of a “flag” word, e.g.

```
#define INITED 1
#define ALIVE 2
#define ZOMBIE 4
#define HUNGRY 8

int status = ...
if (status & HUNGRY) { ... }
```

- Can also pack *n*-bit fields into words in a struct, access them as usual for struct fields

```
struct  _blah {int x:12; int y:12; int z:8};
// x and y are 12 bits, z is 8 bits
```

  - But rare (and horrible).

Other standard libraries

- No time to cover.
- But at least be aware of their existence!
  - assert.h
  - ctype.h
  - errno.h
  - float.h
  - limits.h
  - locale.h
  - math.h
  - setjmp.h
  - signal.h
  - stdarg.h
  - stddef.h
  - stdio.h
  - stdlib.h
  - string.h
  - time.h

See Hill

Addendum: C++

[Not part of the course]

- “The language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates and exception handling, among other features.”
  - Wikipedia
- A vastly more-complex language than C
  - But still compiles to tight efficient code without much OS support
- Detested by language purists
  - Complex, error prone

Stroustrup said it all ...

“C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it
blows your whole leg off”

-- Bjarne Stroustrup, creator/perpetrator of C++

Example 1: Strings

```
#include <string>
#include <iostream>  // I/O streams library, clunky I/O
using namespace std;

int main(void)
{
    string s1 = "Pretty Polly";
    string s2 = "";
    for (size_t i = 0; i < s1.length() - 1; i++) {
        s2 += s1.substr(i, 2);
    }
    cout << s2 << endl;  // Prints s2 followed by end-of-line char
}
```

Prints: `Prreettttyy  PPoolllly`

Example 2: vectors

```
#include <vector>  // vector class gives you dynamic arrays
#include <iostream>
using namespace std;

int main()
{
    int num = 0;
    vector<int> data;    // An empty vector (aka list) of ints
    cin >> num;          // Read an int
    while (num != 42) {  // Read numbers until 42 reached
        data.push_back(num);  // Append to end of list
        cin >> num;
    }

    for (size_t i = 0; i < data.size(); i++) { // Now print them
        cout << data[i] << ' ';
    }
    cout << endl;
}
```

And various other collections classes, including *maps* (aka *dictionaries*)

Example 3: Classes

```
#include <iostream>
using namespace std;

class Vec2d   // A 2D vector with a constructor, and '+' operator
{
  public:
    double x;
    double y;
    Vec2d(double x, double y) // Constructor
    {
        this->x = x;
        this->y = y;
    }
    Vec2d operator+(Vec2d& other) // Define the '+' operator
    {
        return Vec2d(this->x + other.x, this->y + other.y);
    }
};

ostream& operator<<(ostream& out, Vec2d& v)  // Vec2d output operator
{
    out << "(" << v.x << ',' << v.y << ")";
    return out;
}
```

Example 3: Classes (cont'd)

```
int main()
{
    Vec2d v1(10, 20);
    Vec2d v2(20, 30);
    Vec2d v3 = v1 + v2;
    cout << "v1 = " << v1 << ", v2 = " << v2 << ", sum = " << v3;
    cout << endl;  // Line terminator
}
```

And lots of other stuff we don’t have time for …

That's all from me, folks!