Unix and C – historical partners

- Unix (or is it UNIX?) comes in many flavors
  - AT&T Bell Laboratories – System V standard
    - 1969-70: Ken Thompson wrote Unix in "B"
    - 1972: Dennis Ritchie developed C – a better B
    - Unix rewritten in C, 1973 … finally System V, 1983
  - UC Berkeley – BSD standard
    - Started with a copy of System IV, late 1970s
    - Lots of changes/additions in 1980s – now FreeBSD
  - Open source – Linux, since early 1990s
- Many C flavors too – but ANSI standard in 1988

Shared philosophy of C & Unix

- Small is beautiful
  - Each program does just one thing
    - *Pipe* commands (in Unix) or use successive functions (in C) to accomplish more complicated things
  - Less typing is best (using 1970s computers)
    - Short Unix commands (ls, cp, mv, …) and terse C programs
- Users & programmers know what they are doing
  - So brevity is sufficient
  - And very few restrictions (or safety nets) apply

C looks like Java in some ways

- `{ }` – indicates a block, including functions
- Function headers are just like method headers
- Mostly same primitive types – `int`, `double`, …
  - Except C has *no boolean* type – `0` means `false`
  - Also C data sizes can vary, and type conversions are more liberal (i.e., no casts required for "demotions")
  - And C has unsigned integer types
- Same arithmetic/relational operators
  - Including increment/decrement and assignment ops
- Same selection and iteration structures (+ *`goto` !!!*)

Formatted printing to stdout

- `printf(`*`format string`*`, value, value, …);`
  - Almost same as adopted by Java 5
- `%s`, `%d`, `%f` for strings, integers, floating points
  - `printf("my string is %s", stringvar);`
  - `printf("int is %d, float is %f", ivar, fvar);`
- Field width, precision, and more:
  - `printf("int is %5d, float is %8.2f\n", ivar, fvar);`
- See KR chapter 7 and appendix B
- But C programmers are just as likely to process characters *one at a time*
  - See input/output demos at ~cs60/demo01

Constants

- Some are same as Java:
  - `15`, `017`, `0xf` – same value in dec, oct, hex
  - `0.0012`, `1.2e-3` – regular and scientific floats
  - `'c'`, `'\n'` – individual chars; also `"a string"`
- But no `true` or `false` – use non-zero, and `0`
- No `final` keyword – use `const` instead
- Symbolic constants – e.g., `#define MAX 50`
  - *Text substitution* by C preprocessor – more in ch. 4
- Enumerations – e.g., `enum state { in, out };`
  - Type is `enum state` – `in`, `out` are particular *values*

Arrays and character strings

- Declare array and fixed size at same time
  - `int x[50];` /* size must be a constant */
    - Do not use `new` keyword – does not exist in C
    - Also may not reassign array name: `x = …` /* illegal */
- C string: a `char` array, terminated by `'\0'`
  - e.g., `int length(char s[]) {`/* *string length* */

```
        int i;
        for (i = 0; s[i] != '\0'; i++);
        return i;
    } /* note: size of array is probably greater */
```

- Much more coverage of arrays and strings – later
  - And see ~cs60/demo01/longest.c

Function basics

- Must be declared before use
  - Can do with forward declaration (prototype):
    - e.g., `long multiply (int, int);`
    - Parameter names are optional in prototypes
- Must be defined somewhere (for linker)
  - Definition includes header and function body
  - Parameter names are required in definition
    - Parameters are always *copies* of argument values
  - `return` – required if type is not void
    - Value returned is also a copy

Note: old style functions

- No types for parameters
  - Instead – declare types after `)`, before `{`
  - e.g.,
```
double squared(value)
double value;
{ /* function body */ }
```
- Prototypes have empty parameter lists:
  - e.g., `double squared();`
- Problem – compiler cannot verify correct types
  - Old style still works – but recommend do not use
    - See ~cs60/demo01/oldC/

External, `static`, and scope

- External variables: declared outside any function
  - Scope is "global" – whole program can use
    - If `static` – scope is limited to this file
  - Duration is as long as the program is running
  - See calc1.c (K&R pp. 76-79)
- Automatic variables have local scope
  - Includes parameters (local copies)
  - Duration is as long as the function executes
    - If `static` – lasts as long as the program runs
- Note: *all* C functions are external