C philosophy: small is beautiful

- Is underlying philosophy of Unix too
- Each program does just one thing
  - *Pipe* together to do more complicated things
  - Applies at level of C functions too
- Less typing is better than more typing
  - Is why Unix commands are so short – *ls, cp, mv, …*
  - C programs are usually written tersely too
- Users/programmers know what they are doing
  - So brevity works, and few restrictions apply
C data types and variables

- A variable name refers to a memory location
  - Compiler must know the data type stored there
- Just a few basic types (most sizes vary)
  - `char` – 1 byte (8 bits) – number represents a character
  - `int` – for integers
  - `float` and `double` – for floating point numbers
- Also some qualifiers – modify the basic types
  - `short`, `long` – apply to `int` (and `long double` too)
  - `unsigned` – apply to `int` and `char` – positive values
- Must **declare** variable before using it
  - e.g., `int x;` – now can store an integer: `x = 17;`
C constants

- Integers, floats, characters, and C strings:
  - 15, 017, 0xf – same value in dec, oct, hex
  - 0.0012, 1.2e-3 – regular and scientific floats
  - ‘c’, ‘\n’ – individual chars; also “string”

- Symbolic constants – e.g., #define MAX 50
  - Text substitution by C preprocessor – more later
  - New way borrowed from C++: int const MAX = 50;

- Enumerations – e.g., enum state { in, out };
  - Type is enum state – in, out are particular values
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
    - Parameters are always *copies* of argument values
  - `return` – required if type is not `void`
    - Value returned is also a copy
Arrays and character strings

- Declare array and fixed size at same time
  - `int x[50]; /* size must be a constant */`
  - 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 */`

- See character and string processing demo programs in `~mikec/cs12/demo01/`
  - Also shows simple input/output and C program form
Formatted printing to \texttt{stdout}

- \texttt{printf(format, value, value, ...);}
  - \texttt{format} – a string with descriptors for each value

- To print a string variable – use \texttt{\%s} descriptor:
  \texttt{printf("my string is \%s", stringvar);}

- To print a constant string – no descriptors/values
  - Or use \texttt{puts("...");} – prints `\n` at end of string too

- To print an integer (decimal) and a float – \texttt{\%d, \%f}:
  - \texttt{printf("int is \%d, float is \%f", ivar, fvar);}
  - Or describe the field width and/or precision to print:
    \texttt{printf("int is \%5d, float is \%8.2f", ivar, fvar);}

- More \texttt{printf} in KR chapter 7 – and see appendix B
C Pointers

● What are C pointers?
  – Ans: *variables* that store memory *addresses*
    ● i.e., they “point” to memory locations
    ● And they can vary – be assigned a new value
● Background: every variable really has two values

```c
int m = 37; /* What does the compiler do? */
```
  ● (1) sets aside 4 bytes of memory (usually) to hold an `int`
  ● (2) adds `m` and this memory address to a symbol table
  ● (3) stores 37 (one value) in those 4 bytes of memory
  – The other value – a.k.a. `lvalue` – is the memory address
The * has 2 meanings for C pointers
  - (1) to declare a pointer variable:
    ```
    int *p; /* now p can point to an int */
    ```
  - (2) to dereference a pointer:
    ```
    *p = 19; /* stores 19 at location p points to */
    printf("an int value: %d", *p);
    /* finds and prints the value where p is pointing */
    ```

The & retrieves a variable’s lvalue:
```
    p = &m; /* points p at address where m is stored */
    scanf("%d", &m); /* gets an input value for m */
    scanf("%d", p); /* same as above in this case */
```
Pointer types

- Compiler knows type of data a pointer points to
  - For dereferencing, and for pointer arithmetic
- e.g., an `int *` can only point to an `int`
- Exception: a `void *` can point to any type
  - e.g.,
    ```c
    double d = 1.5;
    int x = 6, *ip;
    void *vp = &d; /* vp points to a double */
    vp = &x;    /* okay, now vp points to an int */
    - But cannot dereference `vp` directly – must cast first:
      ```c
      printf("%d", *vp); /* error */
      ip = (int *)vp;  /* now can dereference ip */
    ```
Array names are not pointers (but they are close)

- `int x[10]; /* What does this do? */`
  - Allocates 10 *consecutive* `int` locations
  - *Permanently* associates `x` with the address of the first of these `int` locations – i.e., `x` always points to `x[0]`

- So `&x[i]` is exactly the same as `(x+i)`
  - And `x[i]` is exactly the same as `*(x+i)`

- Also, if `p` is a pointer to `int`, then:
  - `p = &x[0]` is exactly the same as `p = x`
    - But `x = p` is illegal, because `x` is not really a pointer
  - Then `p[i]` is an alias for `x[i]`
  - `++p` moves `p` to point at `x[1]`, and so on