```
CS 214 / 2021-01-27
===================

Reminder: new URL for Webex meetings forthcoming
        look for an announcement; link will be posted to Sakai

No recitations this week
        look for announcements with addresses, links posted to Sakai

----

data in C:
        last time, integers, floats
                chars (fancy integers)
                no bool (make your own)
                no string (just arrays of chars, plus a '\0' terminator)

Literal values
        how we write specific values in source code
        integer literals:
                        decimal: 1, 0, 1000, -15, etc.
                        octal (starts with 0):  010, 0127  <- base 8, not base 10
                        hexadecimal (starts with 0x): 0x123, 0xABCD <- base 16
                suffixes: L (for a long int), U (for unsigned)
                you usually don't need the suffixes, because the compiler will promote values

        situations where you might want to use a suffix:
                the value is too big for a default (signed) int
                        0x12345678  <- 4 byte value (written in hex)
                        0x123456789  <- too big for an int (requires at least 5 bytes)
                        0x123456789L  <- at least 5 bytes (long int is okay for this on iLab)

                you might want to force a promotion
                        int x;
                        long y = x * 100000000;  // possibly a problem if the product is too big
                                                 // to store in an int

                        long y = x * 100000000L; // forces compiler to promote x to a long int
                                                 // and use long int multiplication

                        long y = (long) x * 100000000;  // also "casts" x to long int before multiplying

                these are both pretty rare

        floating-point literals
                0.123, 123.5
                1.23e-18

        character literals  <- actually integers
                'A', 'a', '\n', etc.
                these behave the same as integers
                        'A' - 1

                int x = 'A';   <- sets x to 65  (4 bytes)
                char x = 65;   <- sets x to 'A' (1 byte)

                char x = '\0';  <- sets x to '\0'
                char x = 0;     <- sets x to '\0'

                assume char literals are ASCII characters
                        support for modern text representations is more complicated

        string literals  <-  pointers to constant arrays of chars terminated by '\0'

                char *p = "hello";   <- p refers to a (constant) array of six chars
                printf("%d\n", n);

                        %d - format code, tells printf to print an integer in decimal
                        \n - escape sequence, compiler will replace with newline character

                        printf("%s\n", p);   // prints "hello" followed by a newline

-----

declaring your own types

enums  --  defines a set of named integer values that can be used as constants

        enum direction {left, right, up, down, forward, back};
                <- declares (creates) the type "enum direction"
                <- creates constant values left, right, up, etc.
                        these are just integers
                        by default, left = 0, right = 1, up = 2, etc.

        enum direction input;   <- "input" is a variable that stores a direction

                input = left;   // same as input = 0;

        if (input == up) { ... }

        switch (next_direction) {
                case left:
                        ...
                        break;

                case right:
                        ...
                        break;

                ...
        }

        enum values are not unique (different enums may reuse the same integer represenation)

                enum color {red, green, blue};
                // by default, red = 0, green = 1, blue = 2

                input = red;  // nonsense, but technically possible

                left == red  // true, but you may get a warning from the compiler

        enum values cannot be reused in different types

                enum bad_color { yellow, red, purple}
                                // not allowed, because red is already defined

        you can set your own values, if you want
                enum other_thing { good = 0, bad = 1, awful = 10 };

        enum names are not kept at runtime, so you can't print them as their names
                if you want to print an enum value by name, you have to write your own function
                        printf("%s\n", input)  <- this is a type error
                                <- modern compilers will catch this and report an error
                                <- older compilers will let this through, and then your code will
                                   crash at runtime

                you can also print them as integers

                        printf("%d\n", input);  // prints the numeric value

        why use enums?
                makes your code clearer
                fewer "magic numbers"
                self-documenting code
                        <- says what the expected values are
                some support from type system

        can you do without them?
                sure, just use constants

----

structs <- bundle data into a package

        a struct type has multiple "fields", each field has a name and type

struct rgb_color {
        int redness;
        int greenness;
        int blueness;
        int transparency;
};
        // declares a type "struct rgb_color"
                // bundles together 4 integers
        // declares four field names
                // field names cannot be reused
                // we will see lots of field names with prefixes to avoid name collisions

struct rgb_color background;
        // declares a variable of type struct rgb_color
        // this can be a local variable; the struct value will be stored in the stack
        // i.e., this value is managed by the compiler

struct rgb_color x, y;

        x = y;   <- copies fields from y to x


struct rgb_color background = { 100, 0, 0, 15 };
        // special syntax for initializing struct variables
                // you can only do this when initializing a variable
        // standard only allows you to give fields in order

field access using .

        x.redness = 25;  <- you can set the values of fields
        y.redness = x.blueness;  <- you can access the values of fields

        if (x.redness == y.redness) { ... }

struct fields can be any type, including structs

        struct circle {
                struct point center;
                double radius;
                struct rgb_color fill;  // this stores the actual fields of the color in the circle
        };

        struct circle my_circ;

        my_circ.fill.redness = 235;  // accessing the field of a field

        structs cannot be recursive!

        struct list {
                struct list next;  // not allowed!
                data_t data;
        };
                        // a structure that contains itself would be infinitely large!
                        // this is where we would want to use pointers (coming soon)

----

unions  <-  lets us combine multiple types
        that is, use the same variable/field for different types of data

union int_or_float {
        int inty;
        float floaty;
}

union int_or_float f;
f.inty = 16;

if (f.floaty < 0.0) { ... }

        similar to a struct, except that only one "field" exists at a time
        all the fields are stored in the same/overlapping chunk of memory

        unions do not store which field is currently active!
                there is no way to check at runtime whether f contains a float or an int

        some people use this as a way to sneakily get bit representations of types,
                but not all compilers support this

                this is not the same as casting!

Why would anyone do this?
        to save space
        if we have two variables/fields that we know we won't need at the same time,
                and we only use them in situations where we know which one we have,
                then we can use the same storage for both

        somewhat justifiable for unions of large structs
                lets us fake subtyping; where certain fields are only needed for certain data


        struct circle_data { struct point center; double radius; };
        struct rect_data { struct point topleft; struct point bottomright; }

        struct shape {
                enum {CIRCLE, RECT} type;   // indicate which kind of shape
                union {
                        struct circle_data circ;   // only store data appropriate to that shape
                        struct rect_data rect;
                } data;
        };

        struct shape foo;
        foo.type = CIRCLE;
        foo.data.circ.center = p;
        foo.data.circ.radius = 15;

generally a headache; you will rarely if ever use union

----

arrays  <-  contiguous collection of (same-type) values

declare an array variable using []

        int times[14];  // declares an array containing 14 integers

        array size should be constant (e.g, a number or const)

        array variables are managed by the compiler (you do not allocate/delete them)
        each array variable points to a single unique array


access elements of an array using []

        times[0]  <- first element of array
        times[0] = 14;
        times[1] = times[0] + 2;

"times" by itself can be treated like a pointer (coming soon)

        you cannot assign to the array itself

        array1 = array2;  <- not allowed; you would have to write a loop

multi-dimensional arrays are just arrays of arrays

        int matrix[4][4];  // "matrix" will be 16 contiguous integers

        matrix[0][0]  // access row 0, column 0
        matrix[0]     // refers to the complete first row


        float foo[20][20][20];  // 3-dimensional array

-----

Objects and variables

In C, an "object" is anything that we store in memory

        integers, structs, arrays, even functions

What do we know about an object?
        1. its address (where it is in memory)
        2. its size (how many bytes it occupies)
        3. its type (what sort of data it is)

        <- but only the address is available when your program is running

Variables in C are names for specific objects

        int x;  // create an integer object in memory that I can refer to using the name x

        the variable name/object mapping is fixed / we can't change it

        We can use & to get the address of the variable's object

        &x  <- the location of x's object

Every variable has an associated object
Not every object is associated with a variable
        <- malloc creates objects without variables
```