```
CS 214 / 2021-02-10
===================

Last time:

        malloc, calloc, realloc
        memset
        memcpy, memmove, strcpy, strncpy


#include
#define

#define BUFSIZE 256

        ...
        char buf[BUFSIZE];

        for (i = 0; i < BUFSIZE; ++i) ...

#define MSG "Hello"

#define VARTYPE int
#define VARFMT "%d"

        VARTYPE var;
        printf("Your variable is " VARFMT, var);

                C preprocessor replaces VARFMT with "%d"
                C says that two string literals in a row will be concatenated

        so this is equivalent to:

                int var;
                printf("Your variable is %d", var);

        -> this would allow me to change the type of var and update all my
                format strings
        -> of course, we still have to make sure our definitions are consistent

we can take advantage of conditional compilation to have different definitions
of VARTYPE and VARFMT in different circumstances

conditional compilation: #if, #ifdef, #ifndef, #endif
        -> these allow us to "turn off" or remove chunks of code


#ifdef LONGVAR

#define VARTYPE long
#define VARFMT "%ld"

#else

#define VARTYPE int
#define VARFMT "%d"

#endif

Now, we can use the macro LONGVAR to set the type and format code together


We can define LONGVAR using #define

#define LONGVAR

We can also define some macros on the command line when we call GCC

        gcc -DLONGVAR

                same as #define LONGVAR 1

        gcc -DBUFSIZE=512

                same as #define BUFSIZE 512

#ifdef "if defined"
#ifndef "if not defined"

#if BUFSIZE > 20



macros with arguments
---------------------

idea: put a set of comma-separated arguments in parentheses after the macro name

#define square(X)  ((X) * (X))

        "X" is the argument
                -> within the replacement text, it will be replaced

this will replace
        square(a)

with
        ((a) * (a))


this will replace
        square(x + y)

with
        ((x + y) * (x + y))

<- this is not a function call; it just looks like a function call

        function calls happen when your program is running
        macro substitution occurs before your program is compiled

        arguments to functions are evaluated before the function is called
        arguments to a macro are substituted as-is


#define isupper(C) ((C) >= 'A' && (C) <= 'Z')

        <- safe, can compile efficiently (no function call at run-time)
        <- but watch out if C is an expensive computation or has side-effects

        char c;

        isupper(++c);  <- will increment c twice!

                -> isupper(++c)
                -> ((++c) >= 'A' && (++c) <= 'Z')



#define log(X) log10(X)
        <- quick and dirty rewrite of code to use a different logarithm function

#ifndef WRONGLOG
#define log(X) log10(X)
#endif
        <- do the substitution unless the WRONGLOG macro is set

__FILE__   - replaced by the current file name
__LINE__   - replaced by the line in the source code


#define msg(S) printf("Message %s (%s:%d)\n", S, __FILE__, __LINE__)


        msg("I got here");
        ->
        printf("Message %s (%s:%d)", "I got here", "myfile.c", 1239);


Summary:
        macros are fun and powerful
        ... but don't use them too much
        easy to write obscure, baffling code

        be liberal with parentheses!


separate compilation
--------------------

Header files:
        put function prototypes and type declarations in a single place
                use #include to include header in all files using those functions/types

        #include brings definitions into our source code

Linking
        after compiling separate files, connect names to definitions

Separate compilation

        have multiple .c and .h files
                -> typically a .c file for each standalone concept or feature

separate compiling into two steps
        regular compiling: turn .c file into .o file
        linking: combine multiple .o files into an executable program

        if we make a change in one .c file, we only need to recompile that file
        and then relink
                -> don't need to recompile the entire program
                -> this saves a lot of time for projects with 1000s of source files


managing separate compilation of hundreds/thousands of files is tedious
        use make to do compilation
        make detects which files need to be recompiled
                -> only performs steps necessary to create the requested program


make is very general
        -> not limited to compiling files
        -> not restricted to any particular compiler or language

        we use a "make file" to specify rules that say how to create/build/compile
        the thing that we want

e.g., we might have rules that say
        how to create "demo"
        what other files need to exist to create "demo"
        how to create those files

        by default, make will decide whether it needs to recreate something based
        on the modification times of files
                if a file is older than the files it depends on, then it needs to be
                recreated

next time:
        how to define a make file for fun and profit

        file IO, unless the internet goes down again
```