Day 21 Advanced Compiler Use

Programming with Multiple Source-Code Files

modular programming- divide source code for a single program among 2 or more files

keep some files with general purpose functions use in other programs

like a file that outputs information to screen or takes input from keyboard

each source file is called a **module**

must have a main module with the main() function. other modules called secondary modules. separate header file associated with each secondary module

header files have prototypes for functions in secondary modules, #define directives for symbolic constants and macros, definitons of structures or external variables used in the module

secondary modules have general utility functions- things may want to use in other programs.

ex:

```
gcc program.c keyboard.c keyboard.h
```

will compile and link those programs, the main function **MUST** be in the first program listed. this will create an executable file with the name of program

IF you are recompiling and say keyboard.c was fine and did not need to be recompiled, could simply write:

```
gcc program.c keyboard.obj keyboard.h
```

will use the previously outputted obj file and not go through recompiling keyboard.c again

remember to use **extern** for variables want to use across modules

if you modify a header file must recompile all modules that use it

make utility can handle that with a make file and nmake utility

The C Preprocessor

first compiler component that processes your program

changes source code based on instructions or preprocessor directives in source code

#define- creates symbolic constants and creates macros

substitution macros = symbolic constants

```
#define text1 text2
```

replaces every instance of text1 with text2. if text1 found double quotes, no change

function macros

not case sensitive like little functions

```
#define HALFOF(value) ((value)/2)
```

then in code just put

```
HALFOF(x)
```

and will perform mini-function you defined

when macro parameter is preceeded by # in substitution string, argument converted into a quoted string when expanded:

```
#define OUT(x) printf(#x)
```

so:

```
OUT(Hello World!)
```

expands to:

printf("Hello World!");

concatenation operator- joins 2 strings in macro expressions

```
#define CHOP(x) func ## x
salad = CHOP(3)(q,w);
```

expanded to:

```
salad = func3 (q,w);
```

compilers have settings so can see what preprocessor doing prior to making object file

include directive

preprocessor reads the specified file and inserts it at the location of the directive

angle brackets <> include file in standard directory

"" looks for file in current directory

#if #elif #else #endif

control conditional compilation

can be almost any expression that evaluates to a constant

cannot use sizeof() typecasts, or float

if and endif are required, elif and else are optional

if if evaluates to false compiler evaluates in order the conditions associated with each elif

if nothing is true else compiled

```
#if DEBUG == 1
    debuggin code here
#endif
```

defined()- tests whether a particular name is defined returns true if yes false if no

```
#if !defined( TRUE )
#define TRUE 1
#endif
```

example of how to use this to not include a header file more than once

```
#if defined (prog_h)
#else
#define prog_h
//header file info goes here
#endif
```

#undef- removes definition from a name

Predefined Macros

`__DATE__, __TIME__, __LINE__, __FILE__`

will replace with macros code.

date and time will print current time- good to see when file was actually compiled

using command line arguments

add arguments to your main() function:

```
int main(int argc, char *argv[]) {
```

if write so requires filename, and user does not enter, must have way to handle that