LECTURE 14: MULTI-MODULE PROGRAMS, MAKE (G&A 11.4)

REUSABLE FUNCTIONS

Problem:

We've written a function we'd like to reuse in another program.

Solution 1:

	Creates new problems:
Cut and paste the function into the new program.	Cutting and pasting is tedious.
	If our function changes, we have to update all of our copies.
	Each copy takes up space.

Solution 2:

Separate the function from the original program, compile it separately, link the resulting object module into any other programs we wish to use it in.

We want to create two files:

1. A header file containing the function's prototype.
2. A source code file containing the function itself.

Example: Reverse (G&A, pg. 405)

Header File: reverse.h

```
/* Declare, but do not define, this function */
int reverse(const char[], char[]);
```

Source File: reverse.c

```
#include <stdio.h>
#include "reverse.h"

int reverse (const char[] before, char[] after){
  int i, j, len;
  len = strlen (before);

  for (j = len - 1, i = 0; j >= 0; j--, i++)
    after[i] = before[j];
    after[len] = '\0'; /* terminate reversed string */
}
```

Now we can create a driver file: main1.c

```
#include <stdio.h>
#include "reverse.h"

int main(void){
  char str[100];
  char x[] = "1 2 3 4 5 6 7 8 9";
  reverse(x, str);
  printf("reverse (\"%x\") = %s\n", x, str);
}
```

Note on #include statements:

#include <file>	#include "file"
This variant is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option.	This variant is used for header files of your own program. It searches for a file named file first in the directory containing the current file, then in the quote directories and then the same directories used for <file>. You can prepend directories to the list of quote directories with the -iquote option.

(The C standard says both are implementation-defined, so it could be set up differently depending on your compiler. The above is from GCC's documentation.)

Now you can compile your source files to object modules using the -c option:

```
$ gcc -c main1.c reverse.c
```

You can then link them together in an executable:

```
$ gcc main1.o reverse.o -o main1
```

The advantage here is that you can keep using reverse in new programs, but you don't have to compile it over and over again because it hasn't changed.

Here the savings are trivial, but in a program with a large number of lengthy modules, this can save significant compile time.

MULTI-MODULE PROGRAMS / MAKE

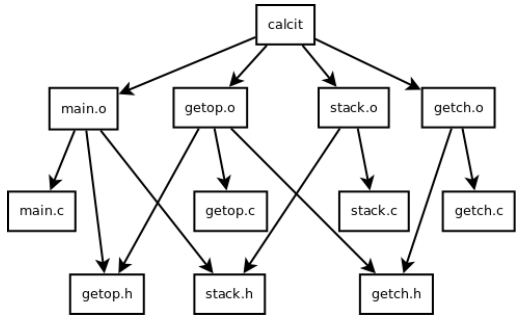
To compile all the .c source files and create an executable:

```
$ gcc main.c getop.c stack.c getch.c -o calcit
```

But if we've already generated object modules for most of our source files and made changes to main.c, we can avoid recompiling:

```
$ gcc main.c getop.o stack.o getch.o -o calcit
```

Dependency Tree



make

For small examples like calcit, the files are not too hard to keep track of, but what if you had a thousand object modules and fifty executable programs?

We can use the make utility to keep track of our dependencies and rebuild our program after changes are made in some subset of its modules without recompiling the entire set.

To run make, you need to first create a makefile.

The default makefile names are GNUmakefile, makefile, or Makefile. If you name your makefile one of these names, you can invoke it simply by typing make at the command line:

```
$ make
```

If you use any other name for your makefile, you must specify the name with the -f option when invoking it:

```
$ make -f fileName
```

makefile rule format

```
target:  dependency list of needed files
  [tab]  command list to create target
```

Note that each command line must begin with a tab character.

Example:

```
calcit:  main.o getop.o stack.o getch.o
         gcc main.o getop.o stack.o getch.o -o calcit

main.o:  main.c getop.h stack.h
         gcc -c main.c
```

Note that in the rule for main.o we are compiling, but not linking.

The order of the rules is important. It must reflect your dependency tree. If make is invoked with a target (e.g., make getop.o) only the target's subtree will be executed. Otherwise it starts with the first rule, so this is where we want the calcit rule.

macros in makefiles

At the top of a make file, you may add macros of the following form:

```
token = replacementText
```

Now, using $(token) in your commands will cause substitution.

Example:

```
CC = gcc
calcit:  main.o getop.o stack.o getch.o
         $(CC) main.o getop.o stack.o getch.o -o calcit
```

When invoking the make utility, you can override the macro:

```
make CC=gccx86
```

last rule: clean

This is your last rule. It removes any existing object files. It is called when you give the command `make clean`.

```
clean:
	rm *.o
```

Clean is not a file, so it does not require a dependency list.

It reduces clutter in the directory by deleting .o files.

implicit build rules

makefile can use implicit (think default) rules to build intermediate files, e.g.:

```
calcit:  main.o getop.o stack.o getch.o
         gcc main.o getop.o stack.o getch.o -o calcit

main.o:  main.c getop.h stack.h

getop.o: getop.c getop.h getch.h

[...]
```

If explicit build rules are not provided, main.o, getop.o, ... will be built using built-in implicit rules.

For C, the implicit rule uses a compiler named CC with options CFLAGS, which you can override at the top of the file, e.g.,

```
CC=gcc
CFLAGS=-m32
calcit:  main.o getop.o stack.o getch.o
         $(CC)$(CFLAGS) main.o [...] getch.o -o calcit

main.o:  main.c getop.h stack.h

getop.o: getop.c getop.h getch.h

[...]
```

You can define new implicit rules with pattern rules: https://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html

LECTURE 14: HEADER FILES, EXTERNAL VARIABLES (K&R § 4.3)

HEADER FILES

A header file uses the extension .h and contains C funtion declarations and macro definitions to be shared between multiple source files.

Each .h file includes all of the function prototypes and symbolic constants needed to invoke those functions.

```
int function(int);
#define VALUE0_FOR_ARGUMENT 0
```

An .h file should never contain a statement that allocates memory!

To use the functions provided by the Math library, you include the header math.h:

```
#include <math.h>
```

The math library is not automatically linked by the compiler like other standard libraries, so if you include math.h, you also need to invoke the compiler with the -lm option (library, math).

Note:	-lm has to be at the end of the compiler invocation.

BACKGROUND: REVERSE POLISH NOTATION (RPN)

For hw4, you need to know about a new type of notation for mathematical expressions.

infix	(123 + 21) * (567 - 432)
RPN	123 21 + 567 432 - *

Note that no parentheses are needed in RPN. Why?

RPN uses a stack:

										432					
				21				567		567		135			
		123		123		144		144		144		144		19440	

RPN CALCULATOR PROGRAM PSEUDOCODE

```
while (next the character is not an EOF)
  if (number)
    push it on the stack
  else if (operator)
    pop operand(s)  /* may be one or two */
    do operation
    push result back on the stack
  else if (newline)
    pop and print value from top of stack
  else error
```

Additional considerations:

operand order	matters for -, /, %
number of operands	unary operators: ~, !

Break down the pseudocode into functions:

- [ ] push/pop
- [ ] get next number or operator (i.e., get op)
- [ ] getch/ungetch

Communication between functions w/ static external

Sometimes closely related functions need to share access to variables in situations where they never call each other.

To solve this problem, we can declare "static" variables inside source file for the functions (before / outside the braces for any of the functions).

Recall: static external variables are available to functions within the file they are declared in only, and their values are preserved in memory (as opposed to automatic variables, which only exist between call and return).

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
int getop(char s[]){
  int i, c;
  while ((s[0] = c = getch()) == ' ' || c == '\t')
    ;
  s[1] = '\0';
  if (!isdigit(c) && c != '.')
    return c;  /* not a number */
  /* collect integer part in string s */
  i = 0;
  if (isdigit(c))
    while (isdigit(s[++i] = c = getch()))
      ;
  /* collect fractional part in string s */
  if (c = = '.')
    while (isdigit(s[++i] = c = getch()))
      ;
  s[i] = '\0';
  if (c != EOF)
    ungetch(c);

  return NUMBER;
}
```

getch.c

```
#define BUFSIZE 100
static char buf[BUFSIZE];  /* buffer for ungetch */
static int bufp = 0;       /* next free position in buf */
/* get a possibly pushed back char from stdin  */
int getch(void){
  return (bufp > 0) ? buf[--bufp] : getchar( );
}
```

ungetch.c

```
/* push char back for getch later */
void ungetch(int c){
  if (bufp >= BUFSIZE)
  printf("ungetch:  too many characters in buffer.\n");
  else
    buf[bufp++] = c;
}
```