C Tutorial - Program Organization

CS 537 - Introduction to Operating Systems

Simple Compilation

- To compile a program in Unix, use *gcc*
- Example:
  - *prompt> gcc myProg.c*
- The previous example compiles an executable file called *a.out*
- To give the executable a different name, use the *-o* option
- Another Example
  - *prompt> gcc myProg.c -o myProg*
- There are lots more command line options
  - see the man pages

File Organization

- Sophisticated programs have multiple files
  - program files (*.c)
    - most of your code goes into these files
  - header files (*.h)
    - mostly prototypes and structure definitions
  - object files
    - compiled program files that are not linked
    - have to be linked to become executable files
  - make file
    - always named *Makefile*
    - responsible for building the entire program

Program Files

- This is what gets compiled
  - includes all of the function definitions
  - before a function can be used, it must be declared
    - prototypes
    - these can either be declared in the program file or in a header file
- Always followed with a *.c* extension

Program File Dependencies

- Possible for a function defined in one program file to be used in another
  - the two (or more) program files must be *linked*
- The simple gcc compilation example shown earlier automatically links your program to some standard libraries
  - that is why you don't need to define *scanf* in your code
- If you want to link to non-standard libraries, you must include them in the compilation command
  - *prompt> gcc myProg.c common.c -o myProg*
    - this compiles both *myProg.c* and *common.c* and links them together

Example

myProg.c

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

int main() {
    int *x, *y;
    x = (int*)malloc(sizeof(int));
    y = (int*)malloc(sizeof(int));
    getNumber(x);
    getNumber(y);
    print("x", *x, "y", *y);
    swap(x, y);
    print("x", *x, "y", *y);

    return 0;
}
```

common.c

```
void getNumber(int* num) {
    printf("Enter a number: ");
    scanf("%d", num);
}

void print(char* s1, int n1,
           char* s2, int n2) {
    printf("%s: %d\n", s1, n1);
    printf("%s: %d\n", s2, n2);
}

void swap(int* n1, int* n2) {
    int tmp = *n1;
    *n1 = *n2;
    *n2 = tmp;
}
```

Header Files

- Often put all the prototypes for a specific program file into a separate file
  - call this the header file
  - always ends in a *.h* extension
- Helps the program files look cleaner
- Allows prototypes of one program file to be easily included in another program file
  - remember, can't use a function until declaring it
- Example

common.h

```
void getNumber(int*, int*);
void print(char*, int, char*, int);
void swap(int*, int*);
```

Object Files

- Instead of compiling multiple files together, they can be compiled separately into object files
  - object files are not executable
  - multiple object files can then be linked to form an executable
  - use the *-c* option of gcc to create object files
- Example
  - *prompt> gcc myProg.c -c*
  - *prompt> gcc common.c -c*
    - these two lines create *myProg.o* and *common.o* object files
  - *prompt> gcc myProg.o common.o -o myProg*
    - links *myProg.o* and *common.o* to create the executable *myProg*

gcc

- The gcc program is both a compiler and a linker
  - it can compile *.c files into object files using the *-c* option
  - it can link multiple object files (*.o) and create an executable
    - *a.out* by default
    - some other name if the *-o* option is used
  - it can both compile and link if multiple *.c files are given on the same line

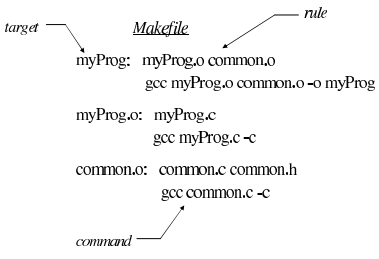
make

- Obviously, for very large programs (with many files), creating and linking all of the object files could be very tedious
- This process can be automated using *make*
- Makefile
  - this is a file that defines exactly how a program should be compiled and linked
  - it only compiles what needs compiling
    - if a file has not been modified since it was last compiled, it won't recompile it
- Whenever you run *make*, it finds the *Makefile* file
  - it then executes all the operations defined in *Makefile*

Makefile

- The basic make file consists of *targets, rules,* and *commands*
  - target: name of the object to be built
  - rules: which files, if modified, would require this target to be re-built
  - commands: how to rebuild a target if any of objects listed in the rules have been changed

Example



More on Makefile

- To help simplify and generalize your make file, variables can be used
  - They are usually declared as all caps
  - They are proceeded by a dollar sign ($) and enclosed in parenthesis when used
- Comments can be included
  - the line must start with a pound sign (#)
  - entire line is ignored

Example Redone

Makefile

```
# compiler
CC = gcc

# linker
LD = gcc

# object files
OBJS = myProg.o common.o

myProg:  $(OBJS)
         $(LD) $(OBJS) -o myProg

myProg.o:  myProg.c
           $(CC) myProg.c -c

common.o:  common.c common.h
           $(CC) common.c -c
```

More on make

- There are million and one options for make
- There are an equal number of uses
- Find a good book or web page to learn more
  - what you have seen here should get you started