Multifile Compilation
CS212 GBA Section

Oftentimes when you make a large program you would like to separate your functions into multiple files for easy organization and portability to other programs you might write. For example, if you are making Mario, you might want to create a file to hold all the functions dealing with the Mario sprite (call it Mario.c) with functions such as run(), jump(), and die(). Perhaps you would also like a file gbaUtilities.c which holds the functions you will use in all/most of your GBA programs (copyOAM for example). And maybe you want a file to deal with the enemies (enemies.c) with functions like addNewEnemy() and moveEnemy() etc. There are multiple ways one could organize this program; you could include all the sprite movements in one file for example. The organization is up to you, provided there is some logical plan.

The compiler will assume these are separate programs and complain about missing functions unless you tell it otherwise. To that effect, you could compile this program as follows

```
gcc Mario.c gbaUtilites.c enemies.c -o Mario.o
```
(-o (file) is the object file you would like to create).

An easy way to compile c programs with gcc is to use what's known as a makefile. There is actually a whole makefile language syntax, we will only cover pieces of it here. You can find more information online if you are curious.

A simple makefile to compile just one .c file would look like this:

```
# NOTE: this file will work for both C and C++ programs.
#  C programs (source code) should end with .c (lowercase)
#  C++ programs (source code) should end with .C (uppercase) or .cc or .cpp
#
# call this file Makefile.  It should be in the current directory you are compiling in
# and type "make myprog" at the unix prompt
#  -- this compiles the file "myprog.c" (or "myprog.C") and creates an
#     executable called "myprog"
#
# to run your code, just type "myprog"
#
# NOTE: if you want more information, do "man gcc" and/or "man make"

CC = gcc        # use the gnu C compiler
CFLAGS = -Wall # Warn about possible problems
.o: .c
      $(CC) $(CFLAGS) -c %.c

##########################################################################
# remove old backup files and core files
clean:
          rm -f *~ core *.o
```

To use this makefile you would type make test.c and test would appear as your executable. This will work in the Windows command line provided your path is pointing to where devkitadv/bin is (the proper Cygwin components were installed along with devkitadv and Windows). It works by default on Unix systems, including Macs.

Multiple files have dependencies (some files depend on others, such as your program depending on some header files). We can tell the makefile about these dependencies (and thus that function moveEnemy() really does exist!) as follows…

```
OBJS = Mario.o gbaUtilities.o enemies.o
Mario: $(OBJS)
   $(CC) -o Mario $(OBJS) $(CFLAGS)
```

The items after the : are the dependencies. In this case, compiling Mario.gba depends on the three object files above.

Making Mario.gba also depends on the header files. Thus, our final makefile for Mario.gba might look like the following:

```
CC=gcc        #the compiler we are using
CFLAGS= -Wall   #our flags: -Wall gives all warnings

MARIO = Mario.o gbaUtilites.o enemies.o        #object files for Mario

MARIOH = gba.h gba_bg.h gba_keys.h gba_regs.h gba_sprites.h gba_types.h
gba_video.h MarioSprite.h                      #header files for Mario

.o: .c $(MARIOH)                               #compiles the .o files w/ headers
       $(CC) $(CFLAGS) -c %.c

Mario:    $(MARIO)                   #links everything together
       $(CC) -o Mario $(MARIO) $(CFLAGS)

clean:                               #cleans up so you can recompile from scratch
       rm -f *.o *~ core
```

It is very important to get the indentation correct here. Tabs are part of the syntax.

A final note: We would highly recommend using a Makefile for your final game. This is a way to ensure that errors are not caused by differential compiling amongst group members, and keeps the file structure coherent.

If you have any questions, feel free to contact a TA.

If you would like more information on Makefiles, there are lots of tutorials online. I happen to like http://www.eng.hawaii.edu/Tutor/Make/ for clarity, though I admit it's a little annoying to flip through the pages.