**OVERVIEW**

The programs you have written so far this semester have all been small enough to put in one file. No more! Real-world programs are much larger and will be written by multiple software developers, thus needing to be split across files. In this lab you will get familiar with this by modularizing your earlier lab on shapes.

**TASKS**

**STEP 0:**

To modularize lab 3, a sensible functional decomposition might have each shape drawn by one function. For example, one such prototype might be:

void drawRectFrame(int width, int height);

Additionally (for pedagogical reasons), you should also have the following function to print a character, and have the draw functions call this (with '\*' as argument) instead of printing directly.

void drawPixel(char c);

Do this for each of the shapes, and place each function in a separate file, named draw{Rect,X,Ltri,Pixel}.C (you only need to do these 3 shapes for this lab).

**STEP 1:**

Write a main function (in lab9.C) that does the following:

1. Output a prompt asking the user to input a character that will be either w, x,  or l  for the 3 shapes.
2. Input a character from the user, and then additional parameters depending on the shape (e.g., the width and height). If the user inputs an illegal character, output an error message and retry.
3. Call the appropriate functon
4. Goto 1. Of course, you're not allowed to use the goto construct, so you should formulate this as a loop.

Do not include prototypes for called functions in this file (see next step).

**STEP 2:**

In this program you only need to specify prototypes for the draw functions in main; however, in general you may need to specify the prototype for some function in multiple other functions/files. Rather than typing the prototype of a function in each of the function's callers, it is better to specify prototypes common to your project in a header (.h) file. To do this, create a file called lab9.h containing prototypes for all common functions. You will need to specify an extern storage class since these are defined in other files. For example, one line might be:

extern void drawRectFrame(int width, int height);

Note that the "extern" is not needed strictly speaking, since that is the default storage class for function prototypes. However,  most consider it good style to explicitly mention.

Since main uses the prototypes in this file, you will also need to have the following at the top of each source file that uses one of these:

#include "lab9.h"

**STEP 2B (OPTIONAL):**

You may wish to make a copy of all .C files you have written so far in a different directory, since its easy to make a typo that overwrites one of these files below.

**STEP 3:**

One way to compile this is (don't do this!):

g++ -o lab9 lab9.C drawRect.C ...

However, that requires you to recompile all files even if there is one minor change, and separate compilation is a better solution. To do this, type:

g++ -o drawRect.o -c drawRect.C

for each source file. The -c option tells the compiler to compile the source into an object file (assembly), but not generate an executable file (since that would require knowledge of all files). The -o option gives the name of the object file - this is not needed since the default for foo.C is foo.o.

After compiling all files, you need to link them and generate an executable file:

g++ -o lab9 drawRect.o ... lab9.C

This will link all the object files and generate an executable file named lab9, which you can run as usual.

You should also check what happens if you fail to include one of the argument object files.

**STEP 4:**

One problem with the above is that it requires the programmer to keep track of dependences between files. For example, if drawrect is changed, then only drawrect and main need be compiled/linked again. The brute force way of repeating all g++ invocations in the previous step is obviously wasteful. This is done using make files to maintain inter-file dependences. .A makefile is named "Makefile" and typically look as follows (note that indentation symbols are required to be tabs, not spaces):

all: program

program: program.cpp program.h

    g++ -Wall -o program program.cpp

clean:

    rm program

The above has 3 rules (all, program, clean). The program rule is read as follows:

The rule is executed if either program.cpp or program.h changes. To execute it the command on the following line is executed.

To execute this make (assuming both Makefile and the source files are in $PWD), simply type "make", which will perform the first rule it finds ("all" in this case).   This rule depends on the program rule, which in turn depends on program.cpp and program.h. Typing "make program" or "make all" would produce the same results, while typing "make clean" would remove the executable file program (be careful not to remove a source file!).

Write a makefile for your program from above that generates an executable file named "lab9".

To check your work (and to see how cool this is), modify drawRectFrame (add a space somewhere) and type make. Note that only drawRect and lab9 are recompiled. Now modify drawPixel (say, to print a dot instead), type make, and note which files are recompiled.

**STEP 5:**

One of the most common ways for distributing multiple code files in the computer science world is as 'tar files'. To do this type:

tar -cvf lab9.tar <list of source files>

Don't forget to include the makefile! The c, v, and f stand for create, verify, and file respectively. To make sure you didn't forget to include a source file, type the above with a -tvf instead of -cvf (t stands for test). You can also do a "man tar" to see all tar options.