Lab 10 - Software Development

Due: end of class

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. Some advantages of this are that it allows for separate development and compilation.

In this lab you will get familiar with this process based on your earlier shapes lab. You will also learn how to create make files.

TASKS

STEP 0:

In Lab 6, you wrote a modular version of the shapes lab (lab 3). First, we will generalize it slightly to draw pixels using arbitrary characters instead of just '*'. To do this, write a function with the following prototype:

```
void drawPixel(char c)
```

Then, make sure each of your functions calls drawPixel instead of printing it directly.

STEP 1:

Put each major function into a separate file. If you have small helper functions you may put them in either a separate file or in the same file as a similar function, but you should at least have one file for each of the shapes and one for main. Do not include prototypes.

STEP 1:

One way to compile the above is to type:

```
g++ -o drawRect.o -c drawRect.cpp
...
```

(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 generating all the object files, you should invoke the linker by typing:

```
g++ -o lab10 drawRect.o ... (include all the object files)
```

But this won't work since there are no prototypes (you followed directions in step 0, right?).

STEP 2:

When writing multi-file programs, it is generally good practice to create header files (.h) that contain all necessary linker information for the corresponding source file. For example, a source file foo.cpp may have a header file foo.h that contains function prototypes and data structure definitions in foo.cpp. Of course, you would only include those prototypes and definitions intended to be accessed from outside foo.cpp.

In this lab, we will create just one header file for all prototypes, though a better practice may be to create one header file per source file. Name this header file main.h, making sure to include all prototypes in here. Make sure to declare the prototypes using an extern storage class since these are defined in other files. For example, one line might be:

```
extern void drawRect(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.

You will also need to include the header file in every file using something declared there:

```
#include "main.h"
```

Now create an executable named lab10, and run to make sure it works!

STEP 2B (optional):

You may wish to make a copy of all source 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 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 looks as follows (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 *program* 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 "lab10".

To check your work (and to see how cool this is), modify one of your draw functions - the command "touch <file>" is one way to modify it without actually changing anything. Note that only that function and main are recompiled. Now modify drawPixel (say, to print a dot instead), type make, and note which files are recompiled.

STEP 4:

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 lab10.tar <list of source files>
```

Don't forget to include the makefile! The c, v, and f stand for create, verify, and file respectively (you can also use z to compress the resulting files). 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.

HIGHLY RECOMMENDED TO LEARN (but not required):

Although not required in this lab, you may want to read about problems caused with multiple definitions and how to resolve them (pp. 489-491).

HAND IN

Your 136 instructor will tell you how to turn in your tar file.

GENERAL COMMENTS FOR ALL PROGRAMS THIS SEMESTER

You should have the following header on all programs:

```
/*
  Author: <name>
  Course: {135,136}
  Instructor: <name>
  Assignment: <title, e.g., "Lab 1">

  This program does ...
*/
```

GRADING

All 135 and 136 programs this semester will be graded on:

- Correctness: Does your program work?
- Testing: Have you generated sufficient and good test data to give reasonable confidence that your program works?
- Structure: Have you structured your code to follow proper software engineering guidelines? This includes readability and maintainability.
- Documentation: How well documented is your code? Good documentation does not repeat the code in English, but explains the point of each code block, highlighting any design decisions and/or tricky implementation details.