Lab 2: Using and creating libraries
CMSC 242
Due: Feb. 5, 2014 by 11:59:59pm

One of the most important skills a programmer develops is the ability to use and create libraries. Libraries allow developers to share code, which not only reduces the time and effort it takes to create software, but also makes it more robust, since it increases the number of people using, testing, and debugging the same code.

In this lab, you will first create a library which provides functions for printing colored reports to the screen using ANSI color codes. In the second part of the lab, you will write a short program that uses the "SDL" library and the “standard math” library. SDL stands for "Simple Directmedia Layer" and is a library used for graphics and sound, especially in games on Linux (although SDL is cross-platform, so SDL programs can also run in Windows or on a Mac). The standard math library is distributed with the “C standard library” and is automatically included with most C compilers.

Finding Documentation

Most libraries have API (Application Programming Interface) documentation that describes, in detail, the functions and data types provided by the library. To write a program that uses the functions provided by the library, you need to be able to read and interpret the API. The API for the standard math library is included in the “Unix Programmer's Manual” which you can access from the command line using the “man” command. One of the functions we will need in this lab is a square root function. Type:

```
            apropos square
```

to search for a suitable function. You will get a list that looks like this:

```
csqrt (3)            - complex square root
csqrtf (3)           - complex square root
csqrtl (3)           - complex square root
rect (6)             - puzzle game based on Divide by Squares
sqrt (3)             - square root function
sqrtf (3)            - square root function
sqrtl (3)            - square root function
```

We don't need complex square roots and we certainly don't need to play “rect”, so the manual pages “sqrt”, “sqrtf”, and “sqrtl” are the most promising ones. To pull up the manual page for the “sqrt” function (on page 3), type:

```
man 3 sqrt
```

You should see a description of the sqrt function from the standard math library. The most important information is in the “SYNOPSIS” section. Notice that it tells us what files to include (math.h). It also gives a prototype for each of the three sqrt functions. The prototype tells us the name of the function, what parameters it takes, and what return type it has. Below the prototype, the man page also tells us to add the “-lm” flag to our compile command.

Most of the functions we will use in this class are documented in the manpages, so you need to be familiar with using them to look things up.

The API documentation for the SDL library is available both in the Unix manpages and on the SDL web site (http://www.libsdl.org). The documentation on the web site is a little easier to use and includes additional information (such as examples and a tutorial) that you can't easily access through the man pages.

Step 1. Creating a Library

Create a folder named Lab2. In that folder, create a file named “color.h”. At the top of your code add C-style comments (using /* and */) that give your name, the course number (CMSC 342), the project number (Lab 2), and the date on separate lines. Then add this code:

```
#ifndef COLOR_H
#define COLOR_H
/* Constants identifying each color */
enum color {NORMAL, BLACK, RED, GREEN, BROWN, BLUE, PURPLE, CYAN, LIGHTGRAY, DARKGRAY, ORANGE, LIGHTGREEN,
            YELLOW, LIGHTBLUE, MAGENTA, LIGHTCYAN, WHITE};

/* Reset the current color to "normal" mode */
void reset();

/* Return the current color */
enum color current_color();

/* Set the color */
void black();
void red();
void green();
void brown();
void blue();
void purple();
void cyan();
void light_gray();
void dark_gray();
void orange();
void light_green();
void yellow();
void light_blue();
void magenta();
void light_cyan();
void white();
#endif
```

This defines the public API for the “color” library. Now, create a file named “color.c” that contains the following code:

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

#define CSI "\e["
static enum color state = NORMAL;

void
reset()
{
        state = NORMAL;
        printf("%s22m%s39m", CSI, CSI);
}

enum color
current_color()
{
        return state;
}

void
black()
{
        state = BLACK;
        printf("%s22m%s30m", CSI, CSI);
}
```

Each of these functions implements one of the prototypes in the header file. We make the “state” variable, which keeps track of the current color private by declaring it *static*. This ensures that the only way to change the state is to call one of the API functions (either reset or one of the color functions). I have given you code for black. Implement the remaining functions using the table below:

Color	First Code	Second Code
Black	22	30
Red	22	31
Green	22	32
Brown	22	33
Blue	22	34
Purple	22	35
Cyan	22	36
Light Gray	22	37
Dark Gray	1	30
Orange	1	31
Light Green	1	32
Yellow	1	33
Light Blue	1	34
Magenta	1	35
Light Cyan	1	36
White	1	37

(The first code turns “high-intensity” off or on. The second code chooses the hue.)

Step 2. Building a library

Most libraries contain more than one module. Let's create a second module that can print a nicely formatted banner around our (colored) text.

Create a file named “banner.h” which contains the following code:

```
#ifndef BANNER_H
#define BANNER_H

void banner(char* message);

#endif
```

This has the usual guards and defines one public function, banner, that takes a C-string and will print that string inside an ASCII-art border.

The implementation should go in a file named “banner.c”:

```
#include "banner.h"

#include <string.h>
#include <stdio.h>

/*
 Print out the string “message” inside an ASCII-art banner.
*/
void
banner(char* message)
{
}
```
Add code for the banner function. If the message string was “Space is fun!”, the banner should look like (note the spaces):

```
*****************
* Space is fun! *
*****************
```

When you are done, the Lab2 folder should have four files: color.h, color.c, banner.h, and banner.c. None of these files has a main function, so we can't create an executable program from them. What we CAN do, is link them together into a library. The first step is to translate them into assembly code object files:

```
gcc -c -fPIC color.c banner.c
```

The -c flag says to stop after compiling and not try to build an entire executable program. The -fPIC flag tells the compiler to build “position independent code” – the kind of code that can be used by a shared library.

We now have two files, color.o and banner.o, that contain each of the modules. To link them together into a single library, we have to generate a “shared object” file. Essentially, this just means concatenating them together and adding a table of contents or “index” that will help other programs find the functions they need. We can do this with a special compile command:

```
gcc -shared -Wl,--soname,libbanner.so -o libbanner.so color.o banner.o
```

The -shared flag says to build a shared library. The -Wl flag is a special command to the linker that tells it the name of our library is libbanner.so – this filename will be embedded into any program linked against the library. The -o flag tells the compiler to output a file named libbanner.so.

You have created a new library! To test it, download the file “test.c” from the course web site:

```
wget http://robert.marmorstein.org/Spring2014/cs242/test.c
```

Then link the test program against your library like this:

```
gcc test.c -L. -lbanner -o Test
LD_LIBRARY_PATH=. ./Test
```

We have to add the LD_LIBRARY_PATH bit and the -L flag because we haven't installed the library to one of the default library folders for the system.

Step 3. Using a library

A simple library like the one we created above can be very powerful if used over and over by many programs. For one thing, the library is “shared” in several different senses. First, the code is reused so you only have to change it in one place to automatically fix bugs in all the programs that link against it – possibly even without recompiling any of the programs that use it. Second, it is shared between processes running on the single computer, so it will be loaded into memory only once. All the programs that use it will share the memory used to store the assembly code instructions. That means that many programs will take less memory.

To see how powerful a library can be, we are going to write a program that uses the SDL library to draw an animated ball that bounces off the four sides of a window.

In the Lab2 folder, create a file named "lab2.c". This file will contain the code we plan to link against the SDL library.

In order to use a library, we have to include the appropriate header files that define the functions we will be using so that the compiler will know how to parse them.

Following your comments in "lab2.c", add the following includes:

```
#include <SDL/SDL.h>
#include <math.h>
#include <stdio.h>
```

These two statements pull in all the functions and data structures we will need for this project. The first include pulls in the SDL library, the second two pull in parts of the C standard library for doing mathematical operations (in particular, square root) and terminal I/O.

Step 4. A main function

Add the usual main function:

```
int main(int argc, char** argv)
{
            return 0;
}
```

We will be adding several functions and global variables to the code, but at this point, you should check that your program compiles and runs. Because we are using additional libraries, you need to add a few flags to the usual compile command. Type:

```
gcc lab2.c -lm -lSDL -o Ball
./Ball
```

Your code should compile and run successfully. Of course, it doesn't do anything yet, but we'll change that soon.

Step 5. Creating a Window

The SDL library has several internal data structures that need to be properly initialized before we call any other functions. To set these up, we call the SDL_Init function. In order to draw on the screen, we also need to create a window of the proper resolution to which we can send drawing commands.

In SDL, a window is just a kind of “drawing surface”. There is a data type “SDL_Surface” that can represent a window, an offscreen buffer, or the graphics framebuffer (in full screen mode). One way to create a window is to use the SDL_SetVideoMode function.

Directly above your main function (well, you should leave a blank line in between), create a function named “initialize_graphics” that takes no parameters and returns a pointer to an SDL_Surface.

Then type:

```
man SDL_Init
```

and read the documentation about the Init function. Using the information in the manual, add code to the “initialize_graphics” function that initializes the SDL subsystems for VIDEO and TIMERS.

Be sure to check for errors. If an error occurs, you should call:

```
exit(-1);
```

to quit your program.

The SDL_Init function allocates memory for SDL's internal data structures. We need to make sure that no matter how the program quits (an error condition, or just normal termination) those data structures will be freed correctly and the video mode set back to normal. SDL provides a function for this called "SDL_Quit". The best way to make sure it always gets called is to use the POSIX function "atexit". Add the following to your initialize_graphics function:

```
atexit(SDL_Quit);
```

This will ensure that whenever the program exits (unless there is a segmentation fault or other crash), the SDL_Quit function will be called.

Now read the documentation for SDL_SetVideoMode. Add the following to the bottom of the "initialize_graphics" function:

```
SDL_Surface* screen = SDL_SetVideoMode(WIDTH, HEIGHT, 32, SDL_SWSURFACE);
if (screen == NULL) {
            fprintf(stderr, "Unable to set %dx%d video: %s\n", WIDTH, HEIGHT, SDL_GetError());
            exit(1);
}

return screen;
```

We are using constants "WIDTH" and "HEIGHT" which we currently haven't defined. At the top of your code (directly after the include statements) add the following defines:

```
#define WIDTH 1024
#define HEIGHT 768
```

This is a reasonable default size for a large window. Add code to your main function that calls initialize_graphics:

```
SDL_Surface* screen;
screen = initialize_graphics();
```

Now recompile and test your code again.

Step 6. Event Loops

Right now the program simply creates a window and then immediately closes it and exits. That's not very interesting. We'd like the program to wait around for a bit. Like most graphics libraries, SDL has functions that let us respond to *events* such as a key press, a mouse click, or the closing of a window. Let's make SDL pause until the user pushes a key. Add the following line to the top of your main function:

```
SDL_Event event;
```

Then at the bottom of the main function, add:

```
while (event.type != SDL_KEYDOWN) {
            SDL_WaitEvent(&event);
}
```

Recompile and test. A window should appear and remain until you press a key.

Step 7. Drawing

One way to draw to the screen is to directly edit the graphics framebuffer. Every SDL_Surface contains an array of pointers that provide direct access to the pixels of the window. We can loop through that array and set the pixels directly to a particular color.

At the top of "lab2.c", before the initialize_graphics function, add a global variable named "white":

```
static Uint32 white;
```

We make it static so that it will be invisible to any other files we link against (including all the files in the SDL library, the standard math library and the standard C library).

Now, to the bottom of the initialize_graphics function add the following code:

```
white = SDL_MapRGB(screen->format, 255, 255, 255);
```

This maps the color with RGB components all set to 255 to a 32-bit integer index, which is stored in the variable "white".

Now create a void function named "clear_screen" that takes a pointer to an SDL_Surface as a parameter. In that function, add the following code:

```
int x, y;
if (SDL_MUSTLOCK(screen)) {
    if (SDL_LockSurface(screen) < 0) {
        return;
    }
}

for (y=0; y < HEIGHT; ++y) {
    for (x=0; x < WIDTH; ++x) {
        Uint32* bufp = (Uint32*)screen->pixels + y*screen->pitch/4 + x;
        *bufp = white;
    }
}

if (SDL_MUSTLOCK(screen)) {
    SDL_UnlockSurface(screen);
}

SDL_UpdateRect(screen, 0, 0, WIDTH, HEIGHT);
```

Because writing directly to the framebuffer is a pretty sensitive operation, SDL requires us to lock the surface before we write to it. This prevents other threads from interfering while we draw. In some circumstances, locking isn't necessary, so SDL provides a macro named "SDL_MUSTLOCK" that lets us avoid unnecessary locking. When we are done, we use SDL_UnlockSurface to unlock the buffer. The heart of this function are the two for loops, which loop through every pixel in the buffer and set it to white. Notice that we are using pointers to directly access the pixels in the surface.

The very last function call tells SDL that we've changed something and it should copy the image that is currently in memory onto the graphics card.

Add code to main that calls "clear_screen". It should go after the graphics have been initialized, but before we enter the event loop.

As usual, compile and test.

Step 8. Drawing a Ball

Drawing directly to the framebuffer is very efficient, but it's awkward and requires locking. SDL provides functions that make drawing shapes and images easier. Unfortunately, none of them are good at drawing circles, so we'll have to hack that together ourselves. (It turns out there is another library, called SDL_draw, that extends SDL with functions for producing circles, lines, and other primitives).

It wouldn't do us much good to draw a white circle on top of a white background, so we need to make another color. Read the man page for “SDL_MapRGB” and use it to create a global Uint32 named “blue” that represents a dark blue color.

At the top of your code, define a new macro named RADIUS with the value 100. Then create a static void function named “draw_circle” that takes four parameters: a pointer to an SDL_Surface named “screen”, an integer named “x”, an integer named “y”, and a Uint32 named “color”.

Inside that function, add this code:

```
int i, j;

static SDL_Rect top_rect;
static SDL_Rect bottom_rect;

top_rect.w = bottom_rect.w = 2;
top_rect.h = bottom_rect.h = 2;

j = -RADIUS;

for (i=x-RADIUS;i<=x+RADIUS;++i)
{
    top_rect.x = bottom_rect.x = i;
    top_rect.y = y-sqrt(RADIUS*RADIUS - j*j);
    bottom_rect.y = y+sqrt(RADIUS*RADIUS - j*j);
    SDL_FillRect(screen, &top_rect, color);
    SDL_FillRect(screen, &bottom_rect, color);
    ++j;
}
```

Then add a call to SDL_UpdateRect.

I hacked this together in just a few minutes, so it has some drawbacks. For one thing, there are some aliasing problems because I didn't use Bresenham's algorithm (which you can learn about if you take CMSC 381: Computer Graphics). Also, I'm turning on each pixel by drawing 1 by 1 rectangles, so if you zoom in, the edges of the circle will look a little blocky. Otherwise, it ought to work okay.

For testing purposes, add:

```
draw_circle(screen, 500,400, blue);
```

to your main function (after clearing the screen, but before the event loop) and recompile.

Step 9. Using Timers

One way to animate the ball is to set off a timer at regular intervals that will trigger the draw_circle function. Every time the timer goes off, we'll erase the old ball, change it's coordinates, and then draw a new ball. Fortunately, SDL has some really nice functions for this (isn't it great not having to write all our own code?)!

At the top of your code, add two new global integer variables named “ball_x” and “ball_y”. They should be static. You should initialize them to 100 and 200, respectively.

Then create a static void function named “move_shapes” which takes a pointer to an SDL_Surface as a parameter. Add this code:

```
/* Erase the old ball */
draw_circle(screen, ball_x, ball_y, white);

++ball_x;
if (ball_x +RADIUS >= WIDTH)
    ball_x = RADIUS;

draw_circle(screen, ball_x, ball_y, blue);

SDL_UpdateRect(screen, 0, 0, WIDTH, HEIGHT);
```

This will move the ball over one pixel. Now all we have to do is trigger “move_shapes” whenever a timer goes off.

SDL uses *callback* functions for this purpose. A callback function is code that you write that SDL timer will trigger when the timer goes off. While we could set up the timer to trigger move_shapes directly, it's better to have a separate function that will trigger move_shapes, because SDL expects callback functions to take exactly two parameters (a Uint32 and a void pointer) and return a Uint32. Add the following code to your lab2.c file:

```
static Uint32 callback(Uint32 interval, void* param)
{
    SDL_Surface* screen;
    screen = (SDL_Surface*) param;
    move_shapes(screen);
    return interval;
}
```

Then, in your main function, add:

```
SDL_TimerID id;
```

to the variable declaration section at the top of the function and:

```
id = SDL_AddTimer(100, callback, screen);
```

just before the event loop.

At the end of the main function, add:

```
SDL_RemoveTimer(id);
SDL_Delay(300);
```

You should also remove the call to “draw_circle” from the main function.

This is a bit of a hack – the timer callback actually runs the move_shapes function in a separate thread. That means that the program could quit in the middle of a drawing function, which would cause the program to crash with a segfault. By turning off the timer and waiting a bit, we give move_shapes enough time to finish so that this doesn't happen.

A better solution would be to use a more elaborate event loop and “event queues”, but that's outside the scope of this lab. If you compile your program, you should now have a ball that slowly floats from left to right and then loops around until you press a key.

Step 10. Better Animation

Modify the move_shapes function so that instead of just moving from left to right, your ball moves diagonally. When it reaches a side of the screen, it should bounce at a perfect 90 degree angle.

My approach to this was to create a “direction” enum that contained the directions “UP_RIGHT”, “UP_LEFT”, “DOWN_RIGHT”, and “DOWN_LEFT”. I would update ball_x and ball_y according to the current direction. If the ball came within RADIUS pixels of a side, I changed to the appropriate direction.

Glitter

For your glitter, I want you to create a file named “glitter.c” that uses your banner library to do something fancy. If you want to add more functions to the color or banner modules, that's fine as long as the old functions still work exactly the same.