12.23.22 Meeting Notes

Topics Covered:

- **Dynamic Memory Management**
  - malloc
  - free
  - calloc
  - reallocarray
- **Setting up a dev environment on Windows & Linux**
  - Setting up an include and lib directory
  - Setting up SDL
- **Using SDL**
  - Creating a window
  - Handling SDL Events
- **Optional C Programming**
  - Bitwise Operators & Bit Flags

DYNAMIC MEMORY MANAGEMENT

Time for the final non-optional topic for C.

Dynamic memory allocation refers to allocating and freeing memory for variables of an unpredictable size. This is largely done with two functions from <stdlib.h>, those being malloc() and free().

malloc() returns a pointer to a newly allocated piece of memory that is any number of bytes long. It takes a size_t value as a parameter, which is just a number of bytes, and it returns NULL on error. You should always check for errors when using malloc because at any moment someone who's using your program could run out of RAM.

```
// Example: allocating memory for a single int and checking for errors

// Allocates memory for an int and returns a pointer to it, returns NULL on error
```

```
int *alloc_int(void)
{
     int *ptr;
     if ((ptr = malloc(sizeof(int))) == NULL)
     {
          fprintf(stderr, "error: couldn't allocate mem
for int\n");
          return NULL;
     }
     return ptr;
}
```

When you're done using the memory you've allocated, pass the pointer to the memory to free() to free the memory back to the operating system. Thankfully, this will never cause an error, so you don't have to do any error handling:

```
free(ptr);
```

It is safe to call free with a NULL pointer, in which case it will do nothing, but make sure you NEVER call free() on a pointer that you did not get from calling malloc() or any other standard library memory allocation function. That will cause undefined behavior.

Also, when your program ends, all memory you've allocated is freed by default, that is, if you're using a competent operating system (so Windows, Mac, Linux... any one people actually use really).

If you want to allocate memory for an array, you can use calloc(), which stands for continuous allocation. This takes a size_t value indicating the number of elements you want to allocate and then another size_t value indicating the size in bytes of each element.

```
int *ptr;
ptr = calloc(4, sizeof(int));

// ptr should now point to the first index of an array
with 4 ints
```

Arrays are not resizable, however, you CAN try using reallocarray on an array to get around this, sort of. reallocarray will find a new memory address for your array that should have enough space for whatever number of elements you want to hold. It has chance of error though, just like malloc and calloc, so remember to check for errors when using it:

```
int *ptr;
ptr = calloc(4, sizeof(int));

// Hmmm now I want an array of size 999
int *new_ptr;
if ((new_ptr = reallocarray(ptr, 999, sizeof(int))) ==
NULL)
{
     free(ptr);
     fprintf(stderr, "frick... don't know how to deal with
this (bad practice don't do this)\n");
     exit(1);
}

// Yay now I have a big array
ptr = new_ptr;
```

I think you should generally avoid reallocarray if you can since, well, it's just cumbersome to use. If you have no other choice though, go for it. If you're making an array that keeps getting values added to it and keeps increasing in size, I'd recommend adding 100 or so elements to it in between reallocarray calls, so that you don't end up calling reallocarray too much. Then, once you're done, call reallocarray one final time to fit the final size of the array.

There's also a function called realloc() that just reallocates a single piece of memory. I haven't ever used it myself though. For more info, just check cppreference.com.

SETTING UP AN INCLUDE AND LIB DIRECTORY

Now that we're finished learning most of C, we can start using SDL, but first we need to learn how to set up C & C++ libraries in general.

C/C++ libraries come in two parts: header files (ending in .h or .hpp) and library files (ending in .a or .so/.dll/.dylib depending on the platform).

Header files, as we know by now, contain function prototypes amongst various other declarations that let the compiler know how to use parts of a library.

Library files are compiled files that contain actual code for library functions. To produce a final executable file, a program called the linker is used to match function prototype calls in our program to their actual function definitions.

Side note: libraries can be linked statically or dynamically. Statically linked programs bundle function definitions from libraries in their final executables and dynamically linked programs depend on external files to access function definitions from libraries. Statically linked programs have no dependencies but are larger in file size. Dynamically linked programs are smaller but depend on external library files to run. You can pass "-static" in your compile command to statically link a program.

Header files are typically kept in a directory named "include" and library files are kept in a directory named "lib".

Side Note: If you're on Linux, which is no one, your package manager should handle this for you by installing C/C++ libraries in the directories /usr/lib and /usr/include. Whenever you compile something, the compiler will automatically search these directories.

If you're on Windows, you have to manage this yourself. I personally like to keep a folder with projects laid out like this:

```
Programming Folder
|____C
     |____include
     |    |____(header file folders here...)
     |____lib
     |    |____(library file folders here...)
     |____bin
     |    |____(stuff you need later here...)
     |
     |____(various project folders here...)
```

To let your compiler know about these folders, you must pass the compiler options "-I<insert include dir here>" and "-L<insert lib dir here>". Example:

> Say I'm in a project directory and my include and lib directories are in the parent folder of my working directory. This should work:
>
> ```
> cc <insert object files> -I../include -L../lib
> -l<insert library name> -o something.exe
> ```

Lastly, as seen in the command above, you need to pass the compiler option(s) "-l<library name>" for every library you're linking against.

Setting up SDL

Now we can finally start using SDL. If I haven't told you guys already, SDL stands for Simple Directmedia Layer. It is a library that provides cross-platform functionality for:

- Creating windows
- Rendering primitives and images
- Getting keyboard, mouse, and controller input
- Playing audio

...among other things. It's not a game engine per se, but we'll be using it to create our own ones.

To set up SDL on Windows, go to https://libsdl.org. Click on SDL Releases and download "SDL2-devel-<version number>-mingw.zip". Extract this zip and enter the folder "x86_64-w64-mingw32" which contains the library files for 64 bit Windows. This folder will contain the include and lib folders. You should copy the contents of these folders to your respective include & lib folders set up in the previous topic.

Also copy the bin folder to your folder. It will contain SDL2.dll which needs to be in the same directory as the executable for any SDL program to run. This is important kind of.

To test your setup, create a new folder in your C folder and write a source file with these contents:

```
#include <stdio.h>

#include <SDL2/SDL.h>

int main(int argc, char **argv)
{
	if (SDL_Init(SDL_INIT_EVERYTHING) < 0)
	{
		fprintf(stderr, "error: failed to initialize SDL. SDL Error: %s\n", SDL_GetError());
		return 1;
	}

	printf("SDL has initialized!\n");

	SDL_Quit();
	return 0;
}
```

So, to recap, SDL_Init() initializes SDL and must be called before any other calls to SDL functions. SDL_Quit() quits SDL and should be called after you've finished using any SDL

functionality you want in your program. When an SDL error occurs, SDL will supply you an error message that you can retrieve a char pointer to by calling SDL_GetError().

For more info on SDL functions check out the SDL API on the SDL Wiki. Whenever I introduce a new function, I'll supply a link to its page on the wiki for you to look through.

Now compile your file with these additional options: "-I../include -L../lib -Wl,-subsystem,console -lmingw32 -lSDL2main -lSDL2"

Now what does all of that (besides the aforementioned -I and -L) do you ask? Well I have no clue. That's just how it works on Windows. On Linux just adding the option "-lSDL2" has the same effect.

Now run your program. If you don't get some weird error message, then congratulations! Your setup works.

CREATING A WINDOW

You can create a window with the SDL_CreateWindow(). This returns a pointer to an SDL_Window struct and it returns NULL on error. Once you're finished using a window, free its memory with SDL_DestroyWindow().

After creating your window, you should call SDL_Delay() to briefly stop your program's execution so the window doesn't close immediately.

Here's a sample program showing off these two functions:

```
#include <stdio.h>
#include <SDL2/SDL.h>
```

```
int main(int argc, char **argv)
{
     if (SDL_Init(SDL_INIT_VIDEO) < 0)
     {
          fprintf(stderr, "sdl failed whoops %s\n",
SDL_GetError());
          return 1;
     }

     SDL_Window *win;
     const char *win_name = "good window";
     const int win_width = 640;
     const int win_height = 480;

     if ((win = SDL_CreateWindow(win_name,
SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, win_width,
win_height, SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE)) == NULL)
     {
          SDL_Quit();
          fprintf(stderr, "window is fail :( %s\n",
SDL_GetError());
          return 1;
     }

     // Wait for 5 seconds
     SDL_Delay(5000);

     SDL_DestroyWindow(win);
     SDL_Quit();
     return 0;
}
```

HANDLING SDL EVENTS

You'll notice that the window from the sample program in the previous topic can't be moved or closed. That's because our program pauses itself with SDL_Delay(), which causes it to ignore all other input.

We can remedy this by creating an event handler. This consists of an SDL_Event struct and a loop that calls SDL_PollEvent until all events have been processed:

```
#include <stdio.h>
#include <stdlib.h>      // For exit()
#include <stdbool.h>
#include <SDL2/SDL.h>

int main(int argc, char **argv)
{
     // Insert SDL initialization and window creation code here

     // Remove your SDL_Delay() call since it's no longer needed
to keep the window from immediately closing

     SDL_Event e;
     while (true)
     {
          while (SDL_PollEvent(&e) != 0)
          {
               switch (e.type)
               {
               case SDL_QUIT:
                    // Insert resource cleanup here (destroying
the window, quitting SDL)
                    exit(0);
                    break;
               }
          }
     }

     // Insert resource cleanup here
     return 0;
}
```

Code Breakdown: SDL_PollEvent() takes a SDL_Event pointer as a parameter and sets the event union's values accordingly to match the type of event that the program receives. This could be multiple things, some examples include:

- Window moving or resizing
- Keyboard input
- Mouse input
- Controller input

To tell what type of event has occurred, we need to check the event union's member variable type. If it is equal to SDL_QUIT, we exit the program, freeing any memory we have allocated before doing so since that's good practice.

An event type of SDL_QUIT signals that the user is trying to close the program, and will occur if you try closing the window. If we were trying to create some kind of weird malware, we could have quit do something other than exiting the program but that would be dumb.

Run this program and see for yourself how the window is now movable and resizable. This is because it's no longer frozen, and instead reacting to events, even if the only ones it actually reacts to are just simply quit events.

Bitwise Operators & Bit Flags

To understand this section, you will need to know conditional expressions, which you might not have learned since I only included them in the notes and forgot to talk about them during meetings. They're explained starting at page 15 of these notes.

C contains the following bitwise operators, which manipulate the bits in numbers:

<< is the bitwise shift left operator. When used on a value, it shifts all of its bits to the left by a specified number of times. New bits appearing on the right will be 0.

```
unsigned char x = 1;
```

```
// In binary, x is 00000001

x = x << 1;

// Now x should be 00000010

x <<= 2;

// Now x should be 00001000

x <<= 99;

// Now x should be 00000000, all of its bits have been
shifted out of it so they are discarded
```

>> is the bitwise shift right operator. Bits that you shift out of the number will not be preserved:

```
unsigned char x = 3;

// x is 00000011

x >>= 2;

// x is 00000000

x <<= 3;

// x is 00000000
```

~ is called “bitwise not” and it inverts all of the bits in a value. Every bit that was 1 becomes 0 and vice versa:

```
unsigned char x = 11;

// x is 00001011

x = ~x;

// x is 11110100
```

`&` is called "bitwise and". It takes two numbers as operands, and evaluates to a number that only holds 1s in the bit positions where both operands hold a 1:

```
unsigned char x = ~11;
unsigned char y = 4;

// x is 11110100
// y is 00000100

unsigned char z = x & y;

// z is 00000100
```

Finally, `|` is called "bitwise or". It is like `&`, but evaluates to a number that holds 1s when either bit at the same position holds a 1.

```
unsigned char x = ~11;
unsigned char y = 11 << 2;

// x is 11110100
// y is 00101100

unsigned char z = x | y;

// z is 11111100
```

Congratulations. You now know all of the bitwise operators in C. Also, note how we used unsigned for our variables in these examples. This is because we don't have to worry about manipulating the sign bit of a number and accidentally changing its value drastically somehow.

Now let's talk about bit flags. They are like boolean values but more space efficient. Boolean values take up an entire byte and use that to only hold a value of true (1) or false (0). Bit flags are values that store a true or false value in each of their bits.

To access the bits in bit flags, we use the | and & operators and singular values that contain a 1 bit in each bit position. Say we wanted to store a player's status ailments in with bit flags, we could set up the following:

```
#define   STATUS_DEAD         (1 << 0)
#define   STATUS_SICK         (1 << 1)
#define   STATUS_POISONED     (1 << 2)
#define   STATUS_HUNGRY       (1 << 3)

int status = 0;
```

To add status ailments, we can |= bits together:

```
status |= STATUS_SICK;
status |= STATUS_HUNGRY | STATUS_DEAD;
```

That was probably a bit too cruel though so we should change it back. We can do that by &= the inverse of the status we want to remove:

```
status &= ~STATUS_DEAD;
```

To check which bit flags are enabled, we can use if statements and &:

```
if (status & STATUS_DEAD)
{
     printf("player died\n");
     printf("oh no :(\n");
}
else if (status & STATUS_HUNGRY)
{
     printf("eat some food dude\n");
}
```

So, yeah, that's how bit flags work. They are memory efficient and also save processing power, since copying over a single value holding multiple bit flags takes less time than copying over multiple individual boolean values. Still though, if you only need to store a single condition, booleans are fine.

Side Note: With this in mind, the expressions we've used in SDL like SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE should make sense now. Those were bit flags that enabled certain parts of SDL we wanted in our programs.