Lab 2.2

Out: 9 February 2005

What you'll learn.

In this lab, you will practice much of what you did in Lab 2.1, but for a slightly more challenging problem. You will be writing code for a technique called "image watermarking." In the printing industry, high quality paper is often "watermarked" with the insignia of the company that manufactures it. Although the watermark is not ordinarily visible, it can be seen by holding the paper up to a bright light.

You will be trying to implement the digital analogue of watermarking. In this case, the idea behind watermarking is to encode additional data in an image without changing its appearance. We will use watermarking to encode a secret message in a picture.

How you'll do it.

You will write two separate programs. One will read in a secret message from the user, open a PPM file, and then encode the message using the watermarking technique. The other program will open an image, extract the secret message, and output it to the terminal.

Task 1: A Makefile to compile your executables

Linking

In Lab 2.1, you compiled a program which contained source code in multiple `.c` files with the command: `gcc -Wall -g -o print_bits print_bits.c ppm.c`

You may notice, however, that this command will always recompile both of your source files even if you have only changed one of them. While this is not problematic for a program with two source files, a larger project can take several minutes to fully compile. In order to solve this problem, gcc allows you to split the process into two steps: compilation and linking. The first step compiles `.c` files into "object files" with the extension `.o`, and is invoked by passing the `-c` flag to `gcc`:

```
gcc -Wall -g -c -o whatever.o whatever.c
```

These "object files" are the binary results of compiling `.c` files, but cannot be run on their own. Multiple `.o` files can be linked together to create an executable on the condition that exactly one of them has a `main()` function. In order to do this, invoke `gcc` with:

```
gcc -o myexecutable file1.o file2.o ...
```

So, in order to compile Lab 2.1 in steps, you would use the following sequence of commands:

```
gcc -Wall -g -c -o print_bits.o print_bits.c
gcc -Wall -g -c -o ppm.o ppm.c
gcc -o print_bits print_bits.o ppm.o
```

Makefiles

At first it appears like compiling a program in several different steps is a terrible idea – you have to type many lines instead of just one. This is where Makefiles come in. A Makefile can automate the process of compiling your program so that you get the benefits of compiling each `.c` file separately, but only have to type `make` in order to completely compile your program.

Makefiles use a kind of scripting language which is specially tailored toward compiling programs. They are extremely powerful, and using them to their full potential requires a lot of practice. For now, we will simply write a Makefile which automatically compiles both our encode and decode programs. We will also give the Makefile the ability to clean up the directory by deleting executables and objects when we don't need them anymore.

A Makefile is divided up into "rules." A Makefile must have at least one rule. Each rule in a Makefile contains a target file, some dependencies, and a sequence of commands that will create the target file from its dependencies. For this lab, our Makefile will contain the following rules:

- `all`: The `all` target is the default target of a Makefile – it will be automatically run if you simply type `make`. Ours will depend on the `encode` and `decode` files so that they both compile when you type `make`.
- `clean`: The `clean` target will remove the executables and object files produced, leaving you with a clean source directory.
- `encode`: The `encode` target will create the `encode` executable. This target depends on `encode.o` and `ppm.o` and will link them to create the executable.
- `decode`: The `decode` target will create the `decode` executable. This target links `decode.o` and `ppm.o` to produce `decode.`
- Implicit Rules: `make` automatically has a rule that any file ending in `.o` depends on its corresponding `.c` file. Additionally, it knows the basic C compiler invocation to compile a `.o` file from its source.

The way that `make` works is to start with the specified target (or `all` if none is specified on the command line) and then recurse into its dependencies until it verifies that they are all "up to date". For `make` to consider a file up to date, it must exist and be newer than the files upon which it depends. Whenever it finds a target that is not up to date, it will execute the sequence of commands associated with that rule. For example, since the `encode` target depends on `ppm.o` and the implicit target for `ppm.o` depends on `ppm.c`, `make` will always make sure that it recompiles `ppm.o` if you've changed its source file. If you've only touched `encode.c`, `ppm.o` will already be up to date and you won't waste time recompiling it.

The syntax to describe a rule in a Makefile is:

```
target: dependency1 dependency2 ...
<TAB> command1
<TAB> command2
<TAB> ...
```

Each rule is separated by one or more completely blank lines.
Note that `make` is *very picky* about the whitespace formatting – you must have one and only one tab before each command for a rule. Using spaces will not function correctly!

Now let's write a Makefile for this Lab. We start with a rule for our `all` target. We simply need to declare that making `all` requires that the `encode decode` targets are up to date:

```
all: encode decode
```

Now we will define a rule for cleaning up our project directory. This target will have no dependencies – even if there are no files to delete, `rm -f` will quietly just exit.

```
clean:
	rm -f *.o encode decode
```

In order to link the executables, we have to have up to date `.o` files. We simply link them together by invoking `gcc` as described above:

```
encode: encode.o ppm.o
	gcc -o encode encode.o ppm.o

decode: decode.o ppm.o
	gcc -o decode decode.o ppm.o
```

This simple `Makefile` functions as is, but lacks the ever-important `-Wall -g` flags that we pass to `gcc`. Additionally, `make` by default invokes the C compile as `cc`, not `gcc`. Even though on our systems, `cc` is actually the same as `gcc`, it's a good habit to invoke it as `gcc`. In order to facilitate these changes, add the following lines to the top of your `Makefile`, above the `all` rule:

```
CC=gcc
CFLAGS=-Wall -g
```

Task: Copy the following files from Lab2.1: `ppm.c`, `ppm.h`. Create two new files called `encode.c` and `decode.c`, and add a `main()` function to both of them.
Construct a `Makefile` according to the instructions above. The filename should be `Makefile` (with a capital 'M'). Test that your `all`, `clean`, `decode`, and `encode` rules function properly.

Task 2: Encode a watermarked image

Here is an overview of how watermarking works. Imagine that we have a large image. Lets say its dimensions are 128×128. An image of that size would have 16,384 pixels, and each pixel is actually three color values ranging from 0 to 255. That means there a total of 49,152 bytes of data in the image (not including the header).

Now suppose we change the least significant bit of one of the color values. The effect on the overall image would be very slight. We are essentially, changing the intensity of a single color channel of a single pixel by a nearly unnoticeable amount. Now suppose we change the least significant bit of every color value in the image. Again, since we are only changing each pixel by a tiny fraction, the resulting image would look identical to the original.

What we have learned is that we can arbitrarily change the least significant bit of any color value without altering the appearance of the image. Remember that there are almost fifty-thousand of these color values in the image. This means that we have fifty-thousand bits of information to play with.

The Encoding Scheme

For any encoding scheme, it is essential to lay out an exact specification for how the data is to be encoded. Our encoding scheme will work as follows:

- The message is a sequence of 8-bit ASCII characters terminated by a null character.
- Each character of the message will be encoded into three consecutive pixels, each of which contains three color values.
- Only the least significant bit for each color value for each pixel will be utilized for the encoding.
- The characters will be encoded starting with the most significant bit.
- Since a character requires eight bits, and three pixels have nine bits that could be utilized, the last bit of the third pixel will remain unused.

Here is a table summarizing how a character will be encoded into three pixels:

pixel - color	bit of the char
1 - red	8 (msb)
1 - green	7
1 - blue	6
2 - red	5
2 - green	4
2 - blue	3
3 - red	2
3 - green	1 (lsb)
3 - blue	unused

Encoding Example

Here is a sample PPM image that is three pixels in width and two pixels in height:

```
P3
3 2
255
200 28 1
198 0 56
81 23 174
16 0 239
212 3 20
46 42 9
```

Currently, decoding this image yields only junk: `#(`

Now, we'll encode a very simple message in the image: the character 'a' followed by a null character. This two character message looks like this in binary: `01100001 00000000`

Here is the PPM image again but now encoded with the simple message. Make sure you carefully examine the difference between the two files so that you understand exactly what is going on:

```
P3
3 2
255
200 29 1
198 0 56
80 23 174
16 0 238
212 2 20
46 42 9
```

Task: Add the following function prototype to `ppm.h`:
`int write128_ppm(FILE* file, image128_t* image);`

Define the corresponding function in `ppm.c`

Begin to write encode.c. The program should first query the user for a file to read from and a file to write to. For now, it should simply read in the PPM from the first file (using `read128_ppm`) and write it back out to the specified output file (using `write128_ppm`. Be sure that both files are properly closed when your program exits.
Use `xv` to verify that your `write128_ppm` function is working properly; the outputted image should be identical to the inputted image.

Task: Add the following function prototype to `ppm.h`:
`int image128_encode(image128_t* image, char* message);`

Define the corresponding function in `ppm.c`

Now, finish writing `encode.c`. Ask the user for a message to encode in the image, and encode the image with this string using `image128_encode` before writing it back to disk. Use `xv` to verify that the images look the same. Use `/course/cs034/bin/decode` to verify that you have properly encoded the string into the image.

Task 2: Decode a watermarked image

Task: Add the following function prototype to `ppm_header.h`:
`int image128_decode(image128_t* image);`

Define the corresponding function in `ppm.c`

Now, finish writing `decode.c`. The program should first query the user for a PPM File. Once the file is open, use `read128_ppm` to bring the image into memory. Then, use `image128_decode` to discover the hidden message and print it out to the terminal. Don't forget to close the file.