Lab 2.1

Out: 9 February 2005

What you'll learn.

In the first part of this lab, you will practice using bitwise operators. In the second part, you will open an image and read formatted data from it. The file format you will be reading is an image format called PPM (Portable PixMap). You can view any .ppm file by typing: `xv <filename>`

How you'll do it.

First, you'll be reusing your code from Lab 1 in order to get input from the user. Remember that a "string" in C is really just an array of `char`s. You'll be printing out the `char`s in binary form, using bitwise operators. In the second part of the lab, you will treat the string of user input as a filename. You will use the `fopen()` function, giving it the user input as the filename. Once you have opened the file, you will read in the data from the file and output it as a binary representation.

Task 1: Print out a binary representation of user input

Data Types

We need to understand exactly how C stores values in memory so that we will know what a string actually looks like in binary. In C, there are several basic data types for representing values. `char` is probably the simplest of all the data types. The length of a `char` is one byte, which is equivalent to eight bits of information. Because it has eight bits, a `char` can take on any of 256 possible values. There is a standardized mapping (called ASCII) of each of these values to one particular letter or symbol. Just as an `int` can be either signed or unsigned, so can a `char`. The most common ASCII symbols fall in the range of 0 to 127, however, there are other symbols in the 128 to 255 range which can be represented only by `unsigned char`s.

Here are a few useful ASCII conversions. These binary representations are each 8 bits long with the least significant bit on the right:

value	binary	character	meaning
0	00000000	'\0'	"null character" – represents the end of a string)
10	00001010	`'\n'`	newline
32	00100000	' '	space
48	00110000	'0'	zero (the digits range from 48 to 57)
65	01000001	'A'	Capital A (capital letters range from 65-90)
97	01100001	'a'	lowercase a (lowercase letters range from 97-122)

Here is a simple string translated into binary:
Hello world.
01001000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100 00101110 00000000

Bitwise Operators

Now that we know what a string should look like in binary, we need to know how to extract each binary bit one at a time. To accomplish this, we will use bitwise operators. Bitwise operators get their name because they apply to each bit of the data individually. When you use ordinary operators, such as the binary operators `+`, `-`, `*`, `/`, and `%`, it is possible to mix and match different data types in a single expression. For example, it is possible to multiply an `int` by a `float` and then place the result in an `int` (this is really done by first converting the `int` to a `float`, multiplying the two `float`s, and then truncating the result back to an `int`). However, when you use bitwise operators, these semantics are thrown out the window. You have direct control over the bits of data. Furthermore, data types are sometimes represented differently depending on which architecture or compiler you are using, so you have to be very careful when using bitwise operators.

Here are some of the commonly used bitwise operators performed on four bit numbers (remember that a `char` is actually eight bits):

```
A = 5 = 0101,  B = 1 = 0001, n = 1
```

operator	meaning	example	result
`<<`	left shift	A << n	1010 = 10
`>>`	right shift	A >> n	0010 = 2
`&`	AND	A & B	0001 = 1
`|`	OR	A | B	0101 = 5
`^`	XOR	A ^ B	0100 = 4
`~`	complement	~B	1110 = 14

Left Shift (`A << n`): The result is equal to `A` shifted `n` bits to the left. This is equal to $A \times 2^n$

Right Shift (`A >> n`): The result is equal to `A` shifted `n` bits to the right. This is equal to $A/2^n$

AND/OR/XOR (`A & B`, `A | B`, `A ^ B`): Each resulting bit is determined by the logical AND/OR/XOR of the corresponding bit in the left and right operands.

Complement (A): This is a unary operator. Each resulting bit is the opposite of the corresponding bit in the operand.

Each of these operators (except complement) can also be used in a compound assignment statement, similar to the `+=` operator. For instance:

`A &= B` is equivalent to `A = A & B`

`A |= B` is equivalent to `A = A | B`

`A <<= B` is equivalent to `A = A << B`

Here are some useful techniques to use in conjunction with bitwise operators. Note that for an unsigned char, n should range from zero to seven:

Creating a mask: A mask is a dummy value with a particular bit or combination of bits set explicitly to one. We will see later how this can be used. Here's how to create a mask with only the n'th bit set to one:

```
mask = 1 << n
```

Setting a bit to one: We can explicitly set a bit to 1 without changing the value of the other bits. Note that if the bit happened to have been set already, this operation has no effect.

```
mask = 1 << n
value |= mask
```

Clearing a bit to zero: We can explicitly clear a bit to zero without changing the value of the other bits. If the bit happened to have already been zero, this operation would have no effect.

```
mask = 1 << n
value &= ~mask
```

Testing a bit: We can also use bitwise operations in conjunction with conditional statements. Here's how to set up a conditional statement that only succeeds if the n'th bit is set to 1.

```
mask = 1 << n
if(value & mask)
  ...
```

Task: Create a file called `print_bits.c`. As you did in Lab1, read in a maximum of 100 characters of input from the user. Make sure that you remove any trailing newline character, and that the string is terminated with a null character (`'\0'`). Now, for each `char` in the string, print out the component bits of the `char`. Make sure that you print the most significant bit first for each character. Adding a space between each set of eight bits will make your output more readable.

Task 2: Open a PPM file, read the data, and print it out in binary.

File Operations

Before we can use a file in any way, we must first "open" the file. The C language provides some standard functions for dealing with files, using the `FILE struct`. There are other methods for dealing with files including file descriptors in UNIX, and file streams in C++, but the `FILE` semantics are very easy to use, and are guaranteed to work with any architecture for which you happen to be compiling. Opening the file gives you a pointer to a `FILE` struct, which you can later use to read or write to the open file. Make sure to always close your open files before exiting the program. What follows is a very brief introduction to the most common file operations. For more information, see the manpages.

```
FILE *fopen(const char *path, const char *mode);
```

Attempts to open the file at the specified path with the permissions specified by mode. Upon success, a pointer to a `FILE` struct is returned, otherwise, the function returns `NULL`[1]. There are several options for mode. The most useful ones are "`r`" (read only) and "`r+`" (read and write).

```
int fclose(FILE *stream);
```

Closes the file that was previously opened using `fopen()`. You should call this function once for each file you open before exiting your program.

```
int fseek(FILE *stream, long offset, int whence);
```

Sets the offset into the file at which subsequent writes/reads will happen. See `man fseek` for info on parameters.

Formatted Files

In its simplest form, a file can be thought of as an array of chars. To make this more useful, a file is often formatted so that discrete values can be extracted from it. You have already seen the `printf()` function. `printf()` is for formatting output so that it is presented to the user in an organized fashion. `printf()` has a cousin called `fprintf()` which operates on files rather than terminal output. [2]

[1] If `fopen()` returns NULL, you can print a nice error message using the `perror()` function. See `man perror` for more info.

[2] In fact, you can also use `fprintf()` to write to the terminal by using a special global `FILE *` called `stdout`. This is what `printf()` does under-the-hood.

```
int fprintf(FILE *stream, const char *format, ...);
```

`fprintf()` takes as parameters a file, a format string, and then some number of values determined by the format string. The values are inserted into the format string. The return value is the number of bytes actually written to the file. The file position is also automatically advanced by this amount.

`[f]printf()` also has a relative that is able to read in from a file based on a specified format string:

```
int fscanf(FILE *stream, const char *format, ...);
```

`fscanf()` uses the same style of formatting string as the `printf()` family of functions. To fully understand how it works, you'll need to read the man page, but here is one example to get you started:

```
#include <stdio.h>

int main() {
  FILE* file = fopen("filename", "r");
  int grade;
  char gradeLetter;
  char name[11];

  fscanf(file, "%10s %d %c", name, &grade, &gradeLetter);
  printf("name = %s\ngrade = %d (%c)\n", name, grade, gradeLetter);

  return 0;
}
```

The code fragment above first opens the file called "filename." Then, it tries to read values from the file in the following order: an arbitrary amount of whitespace (implicit at the beginning of any `scanf` format string), a string having up to 10 characters, more whitespace, an int, more whitespace, and a character.

For example, if the file contains the following string:

```
  wgates   34         F
```

The program will output:

```
name = wgates
grade = 34 (F)
```

PPM File Format

There are literally hundreds of image file formats out there, but we've chosen to work with PPM because it's one of the easiest to parse. Here's an example of a valid PPM file:

```
P3
2 2
255
200 2 1
199 0 56
81 0 174
16 0 239
```

The first two characters of the file must be "`P3`". These two characters serve no purpose other to ensure that we are actually looking at a color PPM file.

The next two fields are integers. The first specifies the width of the image, and the second specifies the height.

The next field is an integer representing the maximum value. In this case, the maximum is 255. Zero is black and white is the maximum value. All the values in between are shades with varying intensity.

Following that is some number of triplets, each representing one pixel. The number of triplets should be width $\times$ height. The pixels are in row-major order. That is, all the pixels in the top row come first, from left to right, then the pixels in the second row, and so on. Each triplet consists of three integers. The first integer is the red channel, the second is the green channel, and the third is the blue. All the color values must fall in the range of $[0, max]$ inclusive, so [0, 255] in this case.

Note that the file above has newlines inserted for clarity. In an actual file, newlines are entirely optional, and are treated as whitespace. Lucky for us, `fscanf()` treats a " " in the formatting string as an undetermined amount of arbitrary whitespace (including newlines), and will automatically skip any whitespace at the beginning of lines.

There are several PPM files for your enjoyment in `/course/cs034/images/`. To view a PPM file:
`xv <filename>`

Structs

A `struct` is a simple construct in C for collecting several basic data types into one concise unit, similar to a Java class without any methods. Often, we define `struct`s to group related data together. `struct`s not only make for cleaner and more organized code, but they allow us to pass these bundles of data around using only a single pointer instead of the many pointers that would be required to pass each piece of data separately. We often put `struct` definitions within a header file so that many C programs can make use of the same kind of `struct` simply by including the corresponding header file.

Here's an example of how to define a new type, called `color_t` as a `struct` that would be useful for keeping track of a single pixel in an image:

```
typedef struct color {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} color_t;
```

The `struct` above is simply a collection of three `unsigned char`s, each one representing one color channel for the pixel. We use `unsigned char`s because we want each color channel to take on the range [0, 255] for representing the intensity of that color in the pixel. Also, there is no need for all of the data members to be of the same type. For example, here is a `struct` for keeping track of the information in an image:

```
typedef struct image128 {
  int width;
  int height;
  color_t pixels[16384];
} image128_t;
```

This `struct` contains an `int` representing the width, another `int` representing the height, and finally a large array of color `struct`s representing the pixels in the image. The number of color `struct`s we are allocating is 16,384, which is just enough for an image of size 128 $\times$ 128. At the moment we will have to limit ourselves to images of this size or smaller. However, in lab 3 you will learn how to get around this issue.

> **Task:** Create a header file called `ppm.h`, remembering to use a header guard. Put the `color_t` and `image128_t struct` definitions in the header file. Also put the following function prototypes in the header file:
> `int read128_ppm(FILE* file, image128_t* image);`
> `void image128_print_bits(image128_t* image);`

The function called `read128_ppm` should take as parameters an open PPM file, and an `image128_t`. The purpose of this function is to read the PPM file and store the contents of the file in memory by filling up the `image128_t struct`. Be sure that the PPM file is valid (it has the correct "magic characters," it has a max of 255, and the image is not too large to fit in the `image128_t struct`). If the image is not valid, the function should return -1, otherwise it should return 0.
Note: `fscanf()` returns the number of items successfully parsed. Check the return value to be sure that the file has a valid format.

The function called `image128_print_bits` should take an `image128_t struct`. Its job is to loop through all the pixels in the `struct` and print out their value in binary to the terminal. For example, a PPM file that looks like:

```
P3
2 2
255
200 2 1
199 0 56
81 0 174
16 0 239
```

would result in the following output to the terminal:

```
11001000 00000010 00000001
11000111 00000000 00111000
01010001 00000000 10101110
00010000 00000000 11101111
```

Task: Create a file called `ppm.c`. Include the header called `ppm.h`. Now, fill in the function bodies for the `read128_ppm` and `image128_print_bits` functions. Remember that you'll also need to `#include <stdio.h>` in order to access `fprintf()` and friends.
To compile your program, type:
`gcc -Wall -g -o print_bits print_bits.c ppm.c`

Task: Create a new `.c` file with a main() to open and read from a PPM file. Treat the user input from the first part as a filename for a PPM file. Use `fopen()` to open the file, and verify that `fopen()` has succeeded. Next, call the `read128_ppm` function that you wrote in order to fill up a `struct` with the data in the file. Next, call your `image128_print_bits` to print out the image in binary. Finally, `fclose()` the file.