CS50

File I/O

Overview

Interacting with data stored on a computer's disk is integral to many programs. Data storage allows information to be used after the lifetime of a program without the hassle of inputting it again. Data is stored on a disk in the form of **files**, collections of data organized in such a way that the computer can understand. Files store a variety of media, including audio, text, movies, pictures, and more.

Key Terms

- file
- file pointer
- file extension
- I/O

Remember that all files are fundamentally just bits – 0s and 1s – that have been organized in a specific way. File types (a "video file," for instance) are merely abstractions of these bits. Likewise, **file extensions**, such as .txt, .c, or .mp3, merely act as references for computer programs that tell them what they should expect to find inside the file and what they should use the file for. So when a computer sees a certain extension, it knows that the corresponding file (i.e., its 0s and 1s) is formatted in a specific way and should be opened with an appropriate program.

Interacting With Files

The "**I/O**" in "file I/O" stands for **input/output**. Thus, file I/O refers to the process of retrieving information from and storing information in files. In general, the file I/O process, broadly, consists of a few steps. First, you must open the file, specifying the ways in which the file can be manipulated. After that, the file's data is free to be manipulated in any of the specified ways. Finally, the file must be closed!

Diving Into Code

Files are interacted with in C via "file pointers," i.e. references to files, which in code are the `FILE *` data type. New file references are usually initialized with the library function `fopen`, which takes two arguments (both strings): the filename and the mode for which the file should be opened.

There are many different ways to manipulate files. The most important of these are reading ("r"), writing ("w"), and appending ("a"), which is just like writing but done directly at the end of the file. To write to a file, we can use `fprintf`, specifying the file pointer to which we want to write and also what it is we want to write. Other functions, like `fgetc`, let us read from a file, getting characters, strings, and the like from the file pointer. Since there are many methods for writing and reading various types of data, we need to make sure to use the one which best suits our needs!

```
1  | // Write to file
2  | FILE *fp = fopen(“document.txt”, “w”);
3  | fprintf(fp, “Hello, world!\n”);
4  | fclose(fp);
5  |
6  | // Read from file
7  | fp = fopen(“document.txt”, “r”);
8  | char c = fgetc(fp);
9  | while (c != EOF)
10 | {
11 |     printf(“%c”, c);
12 |     c = fgetc(fp);
13 | }
14 | fclose(fp);
```

Error Checking

It's also very important to understand and check for potential errors that might occur in the file I/O process. For instance, unlike the previous example, we should always make sure fopen was successful.

Take a look at the code at right and note the error checking to ensure `fopen` was indeed successful. Here, the mode specified was writing, or "w," so failures could occur if the file exists and it is corrupted. Had the mode been reading, or "r" (as in line 7 of the former code), errors could have resulted from trying to read from a nonexistent file. Also, not having the appropriate permission to open a file (to read from or to write to) will also return an error.

```
1 | // Write to file
2 | FILE *fp = fopen(“document.txt”, “w”);
3 | // ensure the file was successfully opened
4 | if (fp == NULL)
5 | {
6 |     fprintf(stderr, “Error opening file.\n”);
7 |     return 1;
8 | }
9 | // continue...
```

CS50

Hexadecimal

Overview

Recall that our computers break everything from ASCII symbols to source code down into combinations of 0s and 1s (**binary**). Those 0s and 1s are not that efficient when it comes to expressing large numbers. To express the decimal number 15, for instance, we need four place values in binary: 1 1 1 1. Because four digits of binary can represent 16 values, computer scientists settled on hexadecimal, a number system of base 16, to represent those larger numbers.

Key Terms

- binary
- hexadecimal
- RGB values

Hexadecimal

In the decimal system (base 10), we have ten digits, 0-9, and each place value represents the next power of 10. So the nth place value can be calculated by taking 10_{n-1}, like in binary (base 2), where we could calculate the nth place value by taking 2_{n-1}.

Similarly, in **hexadecimal** (base 16), we use 0-9 for the first ten digits and the letters A-F for the remaining six. We can think of A as 10, B as 11, and so forth. As you might guess, hexadecimal's place values are based on powers of 16. Note that all the hexadecimal place values are found in binary, albeit more spread out. This makes sense when we remember that 2^4 = 16 and that what takes 4 digits to express in binary can be expressed in 1 digit in hexadecimal.

To convert numbers directly from binary to hexadecimal, simply block off the binary number into chunks of four digits and express what they represent as a single hexadecimal digit. For example, 0 0 0 0 in binary would be a 0 in hexadecimal, and a 1 1 1 1 in binary would be converted into an F (which represents 15) in hexadecimal. This optimization allows us to represent much larger numbers using fewer characters.

Decimal System

3	1	9
100s	**10s**	**1s**
(3 x 100) +	(1 x 10) +	(9 x 1)

300 + 10 + 9
319

Hexadecimal System

1	3	F
256s	**16s**	**1s**
(1 x 256) +	(3 x 16) +	(15 x 1)

256 + 48 + 15
319

Binary System

1	0	0	1	1	1	1	1	1
256s	**128s**	**64s**	**32s**	**16s**	**8s**	**4s**	**2s**	**1s**
(1 x 256) +	(0 x 128) +	(0 x 64) +	(1 x 32) +	(1 x 16) +	(1 x 8) +	(1 x 4) +	(1 x 2) +	(1 x 1)

256+ 0 + 0 + 32 + 16 + 8 + 4 + 2 + 1
319

Hex Colors

One application of the hexadecimal system is the representation of colors. As you may know, all colors are made up of varying levels of red, green, and blue. We refer to these as the **RGB values**. Each of the three colors can have a value between 0 and 255 (16^2-1), which means we need to be able to represent 16,777,216 different colors. And using the hexadecimal number system, we are able to do this in only 6 digits! Imagine using the binary system to express that many colors. It would take 4 times as many digits.

CS50

Images

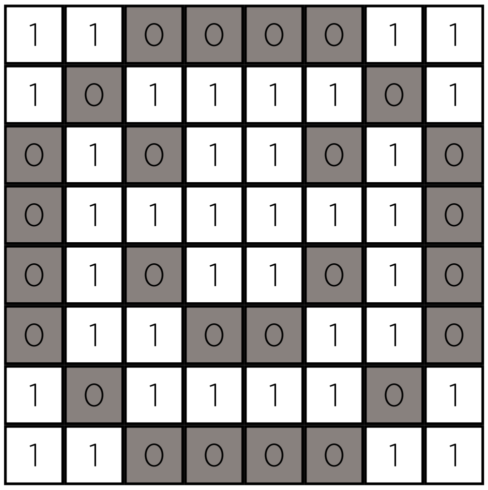
Overview

From social media to cancer screenings, newspapers to comic books, images are important in our lives. Images are stored as files, a series of bytes, just like the programs, word docs, and text files you're used to writing. Common image file formats include bitmaps (.bmp), JPGs (.jpg), PNGs (.png), TIFFs (.tiff), and GIFs (.gif).

Key Terms

- pixel
- hexadecimal
- header
- lossy
- lossless

Bitmaps



Our entire screen is composed of **pixels**, little dots with programmable color and brightness values. A bitmap describes a pattern of all the pixel values that make up an image: when our screen's pixels become those values, the image will appear. A bitmap takes some number of bits per pixel (bpp). More bits can be used to create a more detailed color palette. Back when screens were black and white, just a single bit was used per pixel, with `0 = black`, `1 = white`, as shown at left.

RGB Triples

The bitmaps we've worked with contain three bytes for each pixel in a color image. Each byte specifies a number between `0` and `255`, or, in **hexadecimal**, `0x00` to `0xff`. These three numbers detail how much red, green, and blue to put in that pixel. We can make sense of how this information comes together by thinking about painting, where mixing different combinations of just a few colors can produce many, many different shades. In this case, red, green, and blue can be combined to make the entire rainbow with varying levels of brightness.

Bitmap Headers

Bitmaps also have a **header** – a few bytes at the beginning of the file that tell the display program how to interpret the bits in that file. For the common .bmp Microsoft file extension, we use the struct at right to specify its header. These fields specify the size of the image in bytes, its width and height in pixels, and more. Having this information bundled together ensures that our display program knows exactly how to format the image.

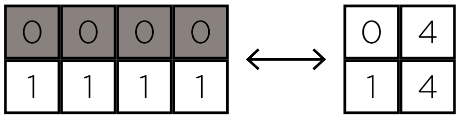
```
typedef struct
{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;

    (...)

} __attribute__((__packed__))
BITMAPINFOHEADER;
```

Other Image File Formats

Do we need to store all of the information corresponding to every pixel in our image file? After all, many images feature a lot of repetition and redundancy (see what we did there?). Is there a way to encode this repetition to create smaller file sizes?

The answer is, typically, yes. Notice the four horizontal repetitions of 0 and 1, respectively, in the example at right. In the file below it, we encoded `(pixel value, repeat number)` pairs. What we've just done here is compressed the original file, resulting in a new file of smaller size.

GIF compression



There are two main types of file compression: lossy and lossless. In **lossy** file compression, files, such as JPGs, are compressed in such a way that data is lost, meaning that the original file cannot be completely recovered. On the other hand, in **lossless** compression, which is used with PNGs and GIFs, no data is lost and the original file can be exactly reconstructed. The compression in the example above falls into this category.

CS50 Structures and Encapsulation

Overview

At a certain point, the usual suspect data types no longer suffice for the kind of work we need to do. Rather, we need to be able to encapsulate data more broadly, allowing us to group related information together. For example, students have names (probably represented by strings), ages (probably represented by integers), and grade-point averages (probably represented by floating-point numbers)--but none of those things matter independently. Instead, all of these things come together and are part of some larger overall entity: the student. Wouldn't it be nice to be able to "bundle" these things together, perhaps allowing us to abstract away some of the underlying specifics? In more modern programming languages, we might do this with a so-called object, but in C, we have a more basic mechanism for this: the **data structure**.

Key Terms

- data structure
- struct
- member

Arrays versus Structs

```
#define STUDENTS 3
string names[STUDENTS];
int classyears[STUDENTS];
float gpas[STUDENTS];
```

```
typedef struct
{
  string name;
  int year;
  float gpa;
}
student;
```

Up until now, if we wanted to group data together, we were limited to an array, each element in which needed to be of the same type. Furthermore, we had to declare the size of the array beforehand. To create a group of variables related to students using arrays, each variable needs to be its own array. And to increase or decrease the amount of students, we need to change the `#define STUDENTS` line accordingly. One advantage of this setup is that, so long as we know the index associated with them, we can directly access every student.

Another way to group data together is with a **struct**. Structs allow us to make new data types out of existing ones. Here we created a type student that has a string, an int, and a float associated with it. We will refer to these as **members**. In this way, we can refer to a specific member of the student type via the line `student.member`, where "member" is the name of whichever member we want to access. A tradeoff of using structs is that we cannot iterate through each field like we can in arrays. In C, arrays are static; so too are the fields in structs. These attributes, such as a name or a year, must be defined. One of the main advantages of storing data in a struct is that we can group data of different types together. Another benefit is that we don't have to declare how many 'students' there will be. Remember that in our earlier implementation of an array, we had to include a **#define** line, but in structs, we can have as many students as we like without having to define that number somewhere in our code.

Implementing Structs

In the lines of code above, we defined a new type called 'student'. Similar to how int is a type, so too is student a type – once we define it. We can now pass variables of type 'student' into functions or into other structs. To create a new a new variable of type 'student', we need to write a line similar to what we would write if we needed to declare a variable of type int: `student s1 = {'Zamyla', 2014, 4.0}`. Now, to access s1's gpa, we can type `s1.gpa`. To pass s1 to a function, let's again look back at how we would do so with ints. For example, `int foo(student x)`, is valid. Similarly, to pass in Zamyla's student information, we can write `foo(s1)`. And if the function takes an argument of type int, we could simply pass in just the year member of s1 by using the line `function(s1.year)`.