

The WAV file format

Lecture 3 WAV and Files

Last lecture we examined how computers represents sound.

This lecture we examine the WAV file format, and discuss how we can create a file in C++. This pave the road for us to actually write a program to create a WAV file.

A specification of Waveform Audio Format (RIFF/WAV) can be found at:

<ftp://ftp.cwi.nl/pub/audio/RIFF-format>

Format highlight:

- The file contains a header followed by some audio data.
- The header identifies the file as a WAV type RIFF file, and then contains a sequence of "chunks"
- The first chunk is the format chunk, which specifies audio format parameters like sampling rate, bits per sample and number of channels.
- The header can also contains chunks that specifies index marks, textual description of the sound, etc, but we will ignore them for our project.
- Normally, data portion may contains raw audio data of 8 or 16 bits.

PMOOP(0396A)

PMOOP(0396A)-3.1

WAV header

The complete WAV file (without special chunks) looks like this:

"RIFF"		packet len		"WAVE"		"fmt "	
chunk len (16)		l	nc	sample rate		data rate	
align	sw	"data"		chunk len			
Audio data							

- nc Number of channels
- sw Sample width: number of bits per sample
- align Number of byte per time

There is a problem: what is actually stored in places like "packet len", where an integer is stored in more than one byte?

PMOOP(0396A)-3.2

PMOOP(0396A)-3.3

Endian

Whenever a number like `0x13579ace` is stored in the memory, we have a problem of representation.

- We will need 4 bytes to store the word. But what should be the first to store?
- **Little-endian**—In some computers (e.g., Intel processors), the least significant byte is normally stored first, i.e. `0xce, 0x9a, 0x57, 0x13`.
- **Big-endian**—In some computers (e.g., Sun Sparc), the most significant byte is normally stored first, i.e. `0x13, 0x57, 0x9a, 0xce`.
- **A WAV file always store in little-endian order**, no matter what machine is used.

This means that if we need to write a program that work for both types of machines, we must be a bit careful.

WAV weirdness: Format of audio samples

In the data chunk stores the audio samples.

- The sample for an earlier time moment is stored completely before the sample for a later time moment.
- If there are two channels, the left channel is stored before the right channel within the same time moment.
- If the data is in 8 bits, the value stored is unsigned, i.e., with value from 0 to 255.
- If the data is in 16 bits, the value stored is signed, i.e., with value from -32768 to 32767 .
- Just like format parameters, 16-bit audio samples are stored in little-endian.

PMOOP(0396A)-3.4

Example sound header

Here is a simple example, which is the beginning of 5772 bytes file `/usr/share/licq/sounds/icq/URL.wav` in Redhat 6.2.

```
00000000: 5249 4646 8416 0000 5741 5645 666d 7420  RIFF...WAVEfmt
00000010: 1000 0000 0100 0100 112b 0000 112b 0000  .....+....
00000020: 0100 0800 6461 7461 5f16 0000 7f7f 7f7f  ....data_.....
00000030: 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  .....
```

- The total RIFF packet size is $0x1684 = 5764$.
- There is only 1 channel.
- Sample rate is $0x2b11 = 11025$ Hz
- Sample width is 8 bits.
- Data chunk size is $0x165f = 5727$.
- The first few sound levels are all `0x7f`, silent middle level.

PMOOP(0396A)-3.5

Difficulty writing WAV file

Note that there are two instances of the WAV file that the size of the WAV data is needed: one for the whole RIFF packet, one for the data chunk.

- That means our program must predict the amount of data we will put into the WAV file.
- Or, the program must store everything in the memory first, and then when the whole file is ready, write everything into the file.
- Or, the program must write everything into the file, and later patch up the file when the file size is known.

Our current program can actually use the first approach, since we just want to write a 1 second sample.

But let's have a bit of foresight. We will use the last approach for our program.

PMOOP(0396A)-3.6

Accessing files in C++

C++ views files like a simple expandable array of characters.

In C++, you read and write a file just like you read something from keyboard and write something to the screen—with some little exceptions.

- File variables are not predefined, since the language do not (and should not) fix the number of files. You have to make the variable yourselves.
- A file can be opened for both reading and writing.
- We usually want to output some strange characters into a file.
- It makes sense to go back to a particular place for files (e.g., to read or write something again, or to read something written earlier), unlike cin and cout.

Unluckily, the full C++ specification of IO library is not really implemented in g++ yet. So stick with the info pages of (iostream).

PMOOP(0396A)-3.7

The ifstream/ofstream/fstream classes

The cin and cout you've been using is actually a istream and ostream type variable. For files, you use variables of other types. These types are defined in the <fstream> header file.

- **ifstream:** Use this if you only need to read a file, not write.
- **ofstream:** Use this if you only need to write to a file, never reading it.
- **fstream:** This can act like either or both ifstream and ofstream.

So you can define a file variable by

```
#include <fstream>
using namespace std;
...
ofstream ofs;
// Now ofs works very much like cout, except it write to a file
```

PMOOP(0396A)-3.8

Open mode

When you open a file, you can (must, for g++ fstream) have an argument that can modify how the file is opened.

ios::in	Open for input. Primarily for ifstream.
ios::out	Open for output. Primarily for ofstream.
ios::trunc	Discard old content. Used with ios::out.
ios::ate	Open and go to the end.
ios::app	Go to the end of file before each write.
ios::nocreate	Never create a file.

You can use logical-or (!) to combine these values together. (You know such operators, right?)

PMOOP(0396A)-3.9

Stream variable and filename

The name of the variable is not the name of the file.

Instead, the variable is associated with an actual file by open:

```
ofstream ofs;
...
ofs.open ("Test.wav"); // Open "Test.wav"
```

Or by adding a parameter when creating the variable:

```
ofstream ofs ("Test.wav"); // Create and open
```

The file is closed when the variable is destroyed. If you want this happen earlier, use this:

```
ofs.close(); // Close it now!
```

PMOOP(0396A)-3.10

Operating on streams

Once a file is opened, we can use the file variable just like cin or cout, e.g., use >> and << operators to read and write the file, use precision or width to set the format, etc. Apart from that,

- ifs.get(c): read the character to c.
- ofs.put(c): write the character c to the stream.
- ifs.read(buf, n): read at most n characters to buf.
- ofs.write(buf, n): write n characters in buf to the stream.

These operations do not treat spaces specially, unlike the >> operator.

Read and write do not try to make the resulting buffer a C-style string. That is, no null character is put into the buffer at the end. This is needed for deal with files that contains the null character itself.

PMOOP(0396A)-3.11

The get and put pointers

As we have said, we will write the file until the end, and then go back to correct a couple of things earlier in the file.

- There is a “put pointer” for each file opened for output. It is the position at which output operations are done.
- We can get the position of the put pointer by using `ofs.tellp()`. It returns a “streampos” type value, which is an integer-type value.
- We can move the put pointer to a particular location `loc` by using `ofs.seekp(loc)`.
- By symmetry, there is also a “get pointer” for files opened for input, and functions called `tellg` and `seekg`.
- In the `g++` library, these functions just manipulate the same pointer. So a `seekg` change also the put pointer.

PMOOP(0396A)-3.12

Trivial example

What will happen for the following simple program?

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    fstream f ("Test.txt", ios::in|ios::out|ios::trunc);
    char c;
    f << "Hello, World!" << endl;
    f.put(67);
    cout << f.tellg () << ' ' << f.tellp () << endl;
    f.seekg (7);
    f.get (c); cout << c << endl;
    f.seekp (4);
    f.put (99);
    f.get (c); cout << c << endl;
}
```

PMOOP(0396A)-3.13

Lecture 4

Creating WAV files

In this lecture we will actually try to write a program to make a WAV file. We will also discuss the problems of this little program, shedding light about how to modify it in the next lecture.

Miscellaneous

We want our program to be usable across systems of different endian conventions. That means no matter whether the machine is itself little endian or big endian, the program should work.

This is one possible ways, although a bit slow.

```
#include <fstream>
#include <cmath>
using namespace std;

void write_lsb (ofstream &st, unsigned int x, int len = 4) {
    for (int i = 0; i < len; ++i) {
        unsigned int r = x & 0xFF;
        x >>= 8;
        st.put(r);
    }
}
```

PMOOP(0396A)

PMOOP(0396A)-4.1

Initializing

Should be easy now.

```
const int sample_rate = 22050, sample_bits = 16, num_channels = 2;
void init_wav_file (ofstream &st, const char *name) {
    st << "RIFF";
    write_lsb (st, 0); // Packet length unknown
    st << "WAVEfmt ";
    write_lsb (st, 16); // Format chunk length
    write_lsb (st, 1, 2); // PCM data
    write_lsb (st, num_channels, 2);
    write_lsb (st, sample_rate);
    write_lsb (st, sample_rate * num_channels * (sample_bits/8));
    write_lsb (st, num_channels * sample_bits / 8.0, 2);
    write_lsb (st, sample_bits, 2);
    st << "data";
    write_lsb (st, 0); // data length
}
```

PMOOP(0396A)-4.2

Writing out a sample in the WAV file

Let us put the scaling logic to the code here. Later we will want to be able to output both 8-bit and 16-bit audio, so we will make it a bit more general.

To avoid losing precision when converting amount number of bits, let's have the `write_sample` function taking floating point argument, and assume that it is of value in the range (-1.0, 1.0).

```
void write_sample (ofstream &st, float amp) {
    int uamp = static_cast<int>(amp / 2 * (1 << sample_bits));
    // Write the sample on all channels
    for (int i = 0; i < num_channels; ++i)
        write_lsb (st, uamp, sample_bits/8);
}
```

Note that we have done one rather funny thing here... do you notice it?

PMOOP(0396A)-4.3

Closing a WAV file

The best time to fix-up the packet length and the data length is probably at the time when you close the file.

Our strategy: just find the put pointer to determine the size of the file. Then we can seek to the header to do the fix-up.

```
const int package_len_pos = 4;
const int data_len_pos = 40;
void close_wav_file (ofstream &st) {
    // Find the file size
    int len = st.tellp ();
    st.seekp (package_len_pos);
    write_lsb (st, len - 8);
    st.seekp (data_len_pos);
    write_lsb (st, len - 44);
    st.close ();
}
```

PMOOP(0396A)-4.4

The main program

Still remember the formula that I want you to calculate? Here it is, except that we will multiply 0.25 to it so that the value cannot exceed our range of (-1.0, 1.0).

```
int main() {
    ofstream st("Test.wav");
    init_wav_file(st, "Test.wav");
    float t = 0;
    for (int i = 0; i < sample_rate; ++i) {
        write_sample (st, sinf (2 * M_PI * t * 1000) * 0.25);
        t += 1.0/sample_rate;
    }
    close_wav_file (st);
}
```

PMOOP(0396A)-4.5

Actually hearing the sound

To actually hear the sound after running the program:

```
play Test.wav
```

But that assume that your Linux computers have a correct sound setup. Try typing `sndconfig` as root and see what happens.

It might correctly configure your system, or suggest installing the ALSA sound driver. It is not too difficult, but require some patience. (Note: if it finds a CS4614/22/24 sound card, don't install version 0.5 ALSA. Instead settle for version 0.4.)

We also want a tool to visualize the wave form. One such program is `xwave`. After installing it, just run the following.

```
xwave Test.wav
```

PMOOP(0396A)-4.6

The reusability problem

This program runs fine, creating a perfect WAV file. It is perfect—if it is the final product that we want.

On the other hand, the program is not really very good if we plan to place it there and say "don't write your own WAV file generator. Use this."

The problem is that for somebody to actually use our code, they must actually find out what they need in our code and do cut-and-paste. They end up needing to understand exactly what our code means.

It is quite possible for another piece of code (e.g., another needed library) to have name clash with our function and variable names. In this case they must do search and replace as well.

It is also very easy to make mistakes in using the code. E.g., any `ofstream` operations can be done on the stream—although they will all cause the WAV generator to misbehave.

PMOOP(0396A)-4.7

Why reusability?

Many tasks are done by more than one program. For example:

- Sort an array
- Solve a set of linear equations
- Build a menu bar
- Parse a HTML or XML document
- Randomize an array
- Playing some music

In fact, many of the tasks that we do will be done again later.

It is of course desirable to design, implement and test once, and then use it forever; rather than doing everything again for every project. This is called software reuse.

PMOOP(0396A)-4.8

Requirements of reusable code

For code to be reused, the following are the minimum requirements:

- It must be easy to find and understand the code.
- It must be reasonably sure that it is correct.
- It must not require separating the code from a larger project.
- It shouldn't require change to adapt to a new program.

This is the minimum requirement for code to be reusable. Note that in this sense, none of the code that you've done so far is reusable. (Bullet 3 and 4 are not satisfied.)

It will be more difficult to program if the result has to be reusable.

PMOOP(0396A)-4.9

Non-technical Obstacles to reusability

There are a lot of non-technical obstacles to reusability. While our course will not focus on them, it is important to know them so that you know when reuse will not work.

- Reusable code programmers are usually not rewarded, usually because project managers care about meeting software deadlines more than reusability.
- Programmers are not usually rewarded when they reuse other code, usually because project managers count productivity by metrics like lines of code.
- Code reuse is usually made impossible by legal constraints.
- Reusable code is usually rendered unmaintained, because the author has completed his project and started working on something else. Unmaintained code is rarely a good target for reuse.

PMOOP(0396A)-4.10

Technical obstacles

On the other hand, even if we finally go for reusable code, there are still significant technical obstacles for reuse. The problem is that we must know what the (usually hypothetical) user want. For example,

- In what platform is the code needed?
- How efficient the code must be?
- How extensible must the code be?

This is not an easy task, and a lot of code is much better not made reusable. However,

- Understanding how code can be made reusable makes it easy to understand the design of many libraries, which are designed to be reused.
- Many projects tends to need some extensibility, and extensibility is just one special form of reusability.

PMOOP(0396A)-4.11