

Lecture 5

Basic Object Oriented Concepts: Library users and writers

Last lecture we created a program which makes a sound file. We also see that it is not good enough for re-use.

In this lecture we will re-try making the program, this time making it a little bit closer to reusable.

By doing this, we introduce the concept of library users and writers, and the differentiation of interface and implementation.

PMOOP(0396A)

PMOOP(0396A)-5.1

Roles of library users and writers

The main objectives of library users and library writers are quite different.

A Library writer tries to make sure that

- library is complete and efficient, hence serves the need of many users.
- library is easy to use, free the users from unnecessary details.
- when the library changes, the user code need few changes.

Library users try to make sure that

- libraries that suit their requirements are chosen.
- using the libraries reduces their programming efforts.
- the libraries are used in a supported way, so that library writers take care of their pattern of usage when changing the library.

PMOOP(0396A)-5.2

PMOOP(0396A)-5.3

Meaning of Encapsulation

Webster 1913

Encapsulation, n. (Physiol.) The act of inclosing in a capsule; the growth of a membrane around (any part) so as to inclose it in a capsule.

Free On-line Dictionary of Computing (foldoc)

encapsulation

...

2. The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.

PMOOP(0396A)-5.4

PMOOP(0396A)-5.5

Separation of library users and library writers

In any programming project, there are two different kind of people involved: the application users and the application programmers. Programmers write code, users run code.

Once we turn our code into a re-usable library, there are two kinds of programmers involved: library writers, and library users. Library writers write a library, and library users use a library to write code for the application user.

For this course, whenever we mention user, we really mean library user—unless specifically qualified as “application users”.

Encapsulation: supporting the differentiation

The C++ support for the differentiation of library users and library writers is in classes.

- A library writer writes classes in his library for the users.
- Library writers can use “**private**” and “**protected**” to specify that part of their program is “implementation details”, and “**public**” to indicate that part of their program is “public interface”.
- The user cannot use data member and member functions marked as private or protected. They are forced to use only the public interface in their programs.
- We call this **encapsulation** (or information hiding): the implementation details of the object is not exposed to the user.
- This free the library writer to change the implementation details without affecting any existing library users.

Example

```
class foo {
public:
    foo(int i): data(i) {};
    int get_data() { return data * data; };
private:
    int data;
};

#include <iostream>
using namespace std;

int main() {
    foo f(5);
    cout << f.get_data() << endl;
    cout << f.data << endl;
}
```

Annotations:

- Written by the library writer (points to the class definition)
- Public interface (points to the public methods)
- Implementation details (points to the private data member)
- Written by the library user (points to the main function)
- Okay: using public interface (points to the correct usage of `f.get_data()`)
- Error: can't use implementation! (points to the incorrect usage of `f.data`)

PMOOP(0396A)-5.4

PMOOP(0396A)-5.5

Another example

In C++, class is the unit of encapsulation.

That means a class can access the internals of any variable of that class, whether or not the variable is *this or not.

In other words, a member function can access the private member of another object of the same class.

```

class Complex {
public:
    Complex(double r, double i): re(r), im(i) {};
    void add(Complex n) { re += n.re; im += n.im; }
private:
    double re, im;
};

```

Okay: private member accessed within class

PMOOP(0396A)-5.6

Class design for WAV file generator

We use a class WAVFile for representing WAV files.

Public interface:

- The user should be able to open and close a WAV file.
- The user should be able to write samples to a WAV file.
- The user should be able to specify the audio parameters: *completeness*.
- The user should be able to get back sampling rate: *completeness*. (To write sample, one must know the sampling rate.)
- The user should be able to not set the audio parameters, or to set only some: *convenience*.

Implementation should be hidden so that library users will not be able to use it: *allow future change*.

PMOOP(0396A)-5.7

The class: overview

```

class WavFile {
public:
    WavFile(const char *name);
    ~WavFile();
    void set_sample_rate(int sr) ...;
    int get_sample_rate() ...;
    void set_sample_bits(int sb) ...;
    void set_num_channels(int n) ...;
    void write_sample(float amp);
    void write_sample(float left_amp, float right_amp);
private:
    ...
};

```

Note that we are using some conventions in selecting names, e.g. always use underscore in method names, Capitalize class names, etc. Again, for *convenience*, so that users don't need to check documentation.

PMOOP(0396A)-5.8

Data invariance: condition to maintain

We can simplify the process to design our class, by making some assumption about the status of the object at times when user code is executed, i.e., when the class writer does not have control.

Such assumptions are called **data invariants**. When we write our code:

- Assume that the data invariants really hold at the entry of the public member functions
- Make sure that they hold again before leaving such functions.
- We say the object to be consistent if the data invariants hold.
- Why we can make the assumptions initially? Because we will set it up when we construct the object: when the constructor is invoked.
- In C++, a destructor will make sure that resources are properly cleaned from a consistent state.

PMOOP(0396A)-5.9

Data invariants of WAVFile

For WAVFile, we will make the following assumptions:

1. A WAVFile object is always associated with a real file.
2. The object is in one of two states:
 - It is "initialized", i.e. the format is chunk written completely and data chunk header is written; or
 - It is "uninitialized", i.e. only the RIFF header is written already.
3. It always stores the audio parameters, which can be changed by the user code when the object is in an uninitialized state.

Member functions can thus access the file directly without checking whether the file is opened already.

PMOOP(0396A)-5.10

Internal representation

Of course, we still have to represent the internal data, and we still have internal member functions. But these are hidden away from the rest of the library user: using **private**.

```

class WavFile {
...
private:
    ofstream st;           // the actual output stream
    bool initialized;
    int sample_rate;
    int sample_bits;
    float sample_offset;
    int num_channels;

    void init();
    inline void write_lsb(unsigned int x, int len = 4);
};

```

PMOOP(0396A)-5.11

Constructor

The constructor is responsible for setting up the data invariant. In particular, it must open the file, and set itself to an uninitialized state. It also set default audio parameters.

```
WavFile::WavFile(const char *name):
    st(name, ios::out|ios::trunc), initialized(false),
    sample_rate(22050), sample_bits(8), sample_offset(1.0),
    num_channels(1)
{
    st << "RIFF";
    write_lsb(0); // Data length
    st << "WAVEfmt ";
    write_lsb(16); // Format chunk length
    write_lsb(1, 2);
}
```

PMOOP(0396A)-5.12

State change

At some time, in particular when there the user code begin writing sample, we have to actually write out the format chunk. We do it with an init call: another implementation details.

```
void WavFile::init()
{
    // Write header
    write_lsb(num_channels, 2);
    write_lsb(sample_rate);
    write_lsb(sample_rate * num_channels * (sample_bits/8));
    write_lsb(num_channels * (sample_bits/8), 2);
    write_lsb(sample_bits, 2);
    st << "data";
    write_lsb(0); // data length
    initialized = true;
}
```

PMOOP(0396A)-5.13

Parameter setting

We also want to allow users to change the parameter. These are supposed to be called when the object is still not initialized.

```
class WavFile {
public:
    void set_sample_rate(int sr) { sample_rate = sr; };
    int set_sample_rate() { return sample_rate; };
    void set_sample_bits(int sb) {
        sample_bits = sb;
        sample_offset = sb <= 8 ? 1.0 : 0.0;
    };
    void set_num_channels(int n) { num_channels = n; };
};
```

There is a problem: what should happen if these are called after initialization? We will deal with this problem later.

PMOOP(0396A)-5.14

Actually writing out samples

This is simple: just generalize our previous code.

```
void WavFile::write_sample(float amp) {
    if (!initialized) init();
    // Write the sample on all channels
    unsigned int uamp = static_cast<unsigned int>
        ((amp + sample_offset) / 2 * (1 << sample_bits));
    for (int i = 0; i < num_channels; ++i)
        write_lsb(uamp, sample_bits/8);
}

inline void WavFile::write_lsb(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)-5.15

Destructor for WAVFile

At some time, when we are convinced that no more operations will be done for the WAVFile object, we need to go back and patch up the header. This can be conveniently done via the destructor.

```
WavFile::~WavFile()
{
    if (!initialized) init();
    // Find the file size
    int len = st.tellp();
    st.seekp(4); write_lsb(len - 8);
    st.seekp(40); write_lsb(len - 44);
}
```

This will be called when the WAVFile variable is "destroyed", i.e. goes out of scope, or is deleted, depending on whether the object is statically allocated or not.

PMOOP(0396A)-5.16

Using the new class

Now, to make a WAV file, we don't need to know exactly what it is. All we need is to create a WAVFile object, and **call its public interface**.

```
int main() {
    WavFile wf("Test.wav");
    int rate = 22050;
    wf.set_sample_rate(rate);
    wf.set_sample_bits(16);
    wf.set_num_channels(2);
    float t = 0;
    for (int i = 0; i < rate; ++i) {
        wf.write_sample(sin(2 * M_PI * t * 1000) * 0.25);
        t += 1.0/rate;
    }
    // Now wf goes out of scope, destructor is called automatically.
}
```

PMOOP(0396A)-5.17

Conclusion

Several points to note:

- The library provides a public interface for the user to do the thing they want: to create a WAV. The user call the public interface without knowing the implementation details.
- In fact, the user code has **no way** to depend on the internal representation of WAVFile. So the library writer can safely change the internal representation without affecting user code.

Exercise

Read the man pages of **open**, **close**, **write** and **lseek**, and rewrite WAVFile using these calls without using fstreams.

This gives you two things: you'll know how Unix really deal with files underneath the library, and also you'll appreciate that the user code need no change under such a drastic change in internal representation.