

What happen on WAVFile error

Lecture 6

Errors and exceptions

In the last lecture we try to build a C++ class that encapsulate the WAV knowledge within a class WAVFile. Using the class, the program don't need to know the details of a WAV file to create one.

There is a major problem in the code: it is not robust. We didn't specify what should happen if something goes wrong. Now we modify the program to handle (most) errors more nicely.

References:

- Reusable C++ book, section 5.3
- Third Edition, section 8.3.

PMOOP(0396A)

PMOOP(0396A)-6.1

The need for error reporting

The problem of error handling:

- In general, library writer can detect an error if they want to. But they don't know what to do with the error: they don't know whether the user want to ignore it, or to correct the problem, or to emit an error message, or to exit the program.
- Doing one of these things unconditionally is unacceptable. E.g., it is not usually acceptable to just quit the program if a file cannot be created: a library user might want to correct the error by asking the application user.
- On the other hand, library users usually know what to do against errors. But since they don't know the implementation details, they don't have a way to detect them!

Instead, the library writers need a way to **communicate the fact of failure** to the library user.

PMOOP(0396A)-6.2

Example: ios error reporting (3rd Ed, Sect 21.3.3)

In ios (the super-class of istream and ostream), there is a "fail" flag that is set whenever there is some error in some error occurs. When this flag is set, all stream operations do nothing.

Therefore, user code should check the flag after each operation:

- For any stream *st*, interpreting *st* as boolean gives false if and only if there has been some error. In this case:
- `st.eof()` indicates that the error has been reading past end of file.
- `st.fail()` indicates that an error has happened, but no character is lost. E.g., trying to read an integer but found non-numeric character.
- `st.bad()` indicates that an error has happened, and some previous writes has lost.
- `st.clear()` clear the flag, so that further I/O can proceed.

PMOOP(0396A)-6.4

Strategies for error handling

The library may deal with errors in many ways:

- It may correct the error. But this applies in very few situations: library writers usually have no idea about what to do about errors.
- It may exit the program immediately.
- It may return an error-code indicating an error has occurred.
- It may flag the object as "erratic".

What the user do depends on what the library writer do.

- It might not need to do anything.
- It might need to check the return value.
- It might need to check the object to see whether it is in an erratic state.

PMOOP(0396A)-6.3

Example

```
int main() {
    int a, b, c;
    cin >> a;
    while (!cin.eof()) {
        cin >> b >> c;
        if (cin) do_work(a, b, c);
        else if (cin.eof()) cerr << "Incomplete last line.\n";
        else if (cin.bad()) {
            cerr << "Unexpected error!\n";
            exit(1);
        } else {
            cerr << "Improper format, skipping line.\n";
            cin.clear(); cin.ignore(INT_MAX, '\n');
        }
    }
    cin >> a;
}
```

PMOOP(0396A)-6.5

Problem of ios-style error reporting

One problem of this approach is of course that it is very troublesome to check every operation. People keep forgetting to check for errors.

But this is not the most serious problem. Let's see the list of problem of this approach:

- Forgetting to check an error leads to incorrect behaviour that goes **completely undetected**.
- To check the error code everywhere makes the user program **very difficult to read**, when the logical flow of the program is obscured by the need to check for errors.
- If the library code has to call the code of the library user, the **library writer might not know what errors can occur**.

What we want is to avoid having to write out most of the checks, in such a way that errors are handled by reporting the problem **by default**.

PMOOP(0396A)-6.6

Exceptions: concepts

C++ has a mechanism that allows this: When an error occurs, we can "throw an exception":

throw exception;

An exception is actually a C++ value. Value of any type can be thrown (as long as it can be copied), including built-in type like int's. But most of the time we want to throw a value of a type that is written solely for throwing.

When an exception is thrown, the program starts **looking for some code that is written for handling the exception**.

It first look at the function that is currently running. If a handler is not found, the function is immediately exited, and the search continues at the point that calls the function.

This repeats until a handler is found, or the main function is exited. In the latter case, a default handler is executed.

PMOOP(0396A)-6.7

Example exception throwing code

```
#include <iostream>
class Complex {
public:
    class ZeroDivide {};
    Complex(double r=0, double i=0): re(r), im(i) {};
    divide(Complex n) {
        if (n.re == 0 && n.im == 0) throw ZeroDivide();
        ...
    }
private:
    double re, im;
};

void f(Complex &a, Complex &b) {
    a.divide(b);
    ...
}
```

PMOOP(0396A)-6.8

Try-catch blocks

One problem remains: how to "handle an exception"?

We use "try-catch" blocks. The **try** block indicates that we expects some error, and the **catch** blocks tell how to handle each type of error.

```
int main() {
    try {
        double r1, i1, r2, i2;
        cin >> r1 >> i1 >> r2 >> i2;
        Complex a(r1, i1), b(r2, i2);
        f(a, b);
    } catch (Complex::ZeroDivide e) {
        cerr << "Division by zero!" << endl;
    } catch (...) { // All other types of exceptions
        ...
    }
}
```

PMOOP(0396A)-6.9

How exception compares to other error handling mechanisms?

Note how exceptions differs from other ways of error reporting:

- By default, an exception is propagated rather than ignored, unlike setting flags or returning error codes.
- Exception does not dictate the strategy to handle the error. The user will determine whether the default is acceptable, and if not catching it. This is unlike, say, exiting the program.
- The user can decide that only some of the exceptions are to be caught, and for others the default applies—unlike setting flags.
- Many statements can share the same exception handlers. For example, if we do a lot of Complex operation and want to return a special value in case of numeric error, we enclose the whole thing within one large try block. This is unlike setting flags.

PMOOP(0396A)-6.10

Adding exceptions to WAVFile

```
class SoundError {
public:
    SoundError(string s): reason(s) {};
    string reason;
};

WavFile::WavFile(const char *name):
    st(name, ios::binary), initialized(false),
    sample_rate(22050), sample_bits(8), sample_offset(1.0),
    num_channels(1)
{
    if (!st) throw SoundError("Cannot open file");
    st << "RIFF\0\0\0\0WAVEfmt ";
    write_lsb(16); // Format chunk length
    write_lsb(1, 2);
    if (!st) throw SoundError("Write failed");
}
```

PMOOP(0396A)-6.11

Changes for the main program

```
int main() {
    try {
        WavFile wf("Test.wav");
        int rate = 22050;
        wf.set_sample_rate(rate);
        wf.set_sample_bits(16);
        wf.set_num_channels(2);
        for (int i = 0; i < rate; ++i) {
            float angle = i / (rate/1000.0) * M_PI;
            wf.write_sample(sin(angle) * 0.25);
        }
    } catch (SoundError e) {
        cerr << "Sound Error: " << e.reason << "." << endl;
    }
}
```

PMOOP(0396A)-6.12

Conclusion and Exercise

Short summary

- Library writers can detect errors but can't in general handle them. Library users are in the reverse situation.
- Exception provides a clean mechanism for library writers to report that an error has occurred, so that users can handle them.
- Upon receiving an exception, by default the current function is terminated and the exception is forwarded to caller.
- If a function knows how to handle an error, it will establish a try-catch block to handle the error.

Exercise

- With exceptions, cleanly detect and handle the case if the user tries to change the audio parameter after initialization. Modify main() to write out a warning message and continue working.

PMOOP(0396A)-6.13

Lecture 7

More on exceptions, and exception safety

Last lecture we start our examination of the C++ exception facilities.

In this lecture we examine how exceptions interacts with the remainder of the programming environment.

We will also see that exception introduce non-local control flow, the problems that this causes, and the way to handle them.

- Third Edition, chapter 14.
- Reusable C++ book, section 5.5

PMOOP(0396A)

Why iostream don't use exception?

If exceptions are this nice, why don't iostream throw exceptions?

The reason turns out to be historic: when iostream is designed, there is no exception in C++.

When exceptions really come in, there are already a lot of programs that depend on its error handling mechanism that predate exceptions.

A sudden change of the public interface from setting an internal state to throwing an exception would cause a lot of code to fail.

But even though that is the case, exceptions actually get into the new C++ standard, although most compilers didn't support that yet, and it needs to be explicitly enabled.

PMOOP(0396A)-7.1

Exceptions in iostream (not in g++ yet, 3rd Ed Sect. 21.3.6)

Under the new standard, each stream has an "exception state" associated with it. What will trigger an exception depends on this state.

It can be the bitwise-or of any combination of the values **ios::badbit**, **ios::failbit** and **ios::eofbit**, indicating that exceptions should be thrown when the stream becomes bad, fails or reaches EOF.

You can set and get the exception state by using the member function *exceptions* of the stream:

- `st.exceptions(flag)`: set the exception flag of the stream *st* to *flag*.
- `st.exceptions()`: get the current exception flags.

Example: `cin.exceptions(ios::badbit | ios::failbit)` asks cin to throw exception on both bad and fail, but not on eof.

PMOOP(0396A)-7.2

Example

```
// Not supported by g++ currently :-{
int main() {
    cin.exceptions(ios::badbit); // Really don't expect cin going bad
    int a, b, c;
    cin >> a;
    while (!cin.eof()) {
        cin >> b >> c;
        if (cin) do_work(a, b, c);
        else if (cin.eof()) cerr << "Incomplete last line.\n";
        else {
            cerr << "Improper format, skipping line.\n";
            cin.clear(); cin.ignore(INT_MAX, '\n');
        }
        cin >> a;
    }
}
```

PMOOP(0396A)-7.3

Where exceptions are bad

Note that in the above code, we do expect that EOF will be reached at the end. In this case, it is not really nice to have it throwing exceptions:

```
int main() { ... // very ugly use of exception
  try {
    cin >> a;
    while (true) {
      try { cin >> b >> c; }
      catch (...) {
        if (cin.eof()) { cerr << "Incomplete last line.\n"; exit(1); }
        else throw;
      } ...
      cin >> a;
    }
  } catch (...) {
    if (!cin.eof()) cerr << "Improper format, skipping line.\n";
  }
}
```

PMOOP(0396A)-7.4

Re-throwing exceptions; Scope of exception handler

Re-throwing

- In some cases (e.g., the code of the last page), we might want to catch an exception but can handle only some cases.
- If we just want to rethrow the caught exception, just say **throw** suffices.

Scope of exception

- When an exception is thrown, everything inside the try block that catches the exception is cleaned up.
- That means the exception handler cannot use the variables within it: they no longer exist.

PMOOP(0396A)-7.5

Example on exception scoping

In the exception handler below, b and c are gone before the handler executes. So they cannot be accessed by the handler, although the handler can be reached only after b and c are assigned values.

```
void f() {
  int a = 10;
  try {
    int b = 0, c = 0;
    b = do_something(a); // do_something can throw exception
    c = do_something(b);
  } catch (...) {
    cout << "Exception caught!" << endl;
    cout << "a is " << a << endl; // Okay: a is still there.
    cout << "b is " << b << endl; // Oops: no b in scope!
  }
}
```

PMOOP(0396A)-7.6

Exception safety

Since some function might throw exceptions, the control flow of linear code might be interrupted. This can cause quite some problems.

- You might have temporarily set some values in your objects, violating the data invariance. You expect to reestablish the invariant in some code later, but an exception occurs.
- You might have allocated some object, expecting to clean it up in some later code, but that code is never executed due to an exception.

A piece of code is said to be exception safe if it behaves correctly even if exceptions are thrown (and possibly caught).

Note that **statically allocated objects will be cleaned up** when an exception is thrown, i.e., they will have their destructor executed. (Not for Java.)

The real allocation problem occurs only for those dynamically allocated objects (those allocated by **new**). We will see one example soon.

PMOOP(0396A)-7.7

Example

Suppose for the moment that write_1sb could throw exception. What would happen if the 4th write_1sb throw one? Is the object still consistent?

```
void WavFile::init()
{
  // Write header
  write_1sb(num_channels, 2);
  write_1sb(sample_rate);
  write_1sb(sample_rate * num_channels * (sample_bits/8));
  write_1sb(num_channels * (sample_bits/8), 2);
  write_1sb(sample_bits, 2);
  st << "data";
  write_1sb(0); // data length
  if (!st) throw SoundError();
  initialized = true;
}
```

PMOOP(0396A)-7.8

Exception-safe version

It is usually difficult to keep exception safety. But this time we can:

```
void WavFile::init()
{
  int oldp = st.tellp();
  try {
    write_1sb(num_channels, 2);
    ...
    write_1sb(0); // data length
  } catch (...) {
    st.clear(); // Clear the fail bit
    st.seekp(oldp); // Restore the original state
    throw; // Rethrow it: something bad happened
  }
  initialized = true;
}
```

PMOOP(0396A)-7.9

Resource leak with exceptions

As we have noted, resource leak can occur with dynamically allocated objects. For example, suppose our `main()` looks like this:

```
int main() {
    // We usually need this to allow polymorphism (week 4)
    WavFile *wf = new WavFile("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;
    }
    delete wf;
}
```

This might not be executed whether or not we catch the exception: no one will close the file!!!

PMOOP(0396A)-7.10

An ugly fix

Knowing the problem, we can fix it like this:

```
int main() {
    // We usually need this to allow polymorphism (week 4)
    WavFile *wf = new WavFile("Test.wav");
    try {
        ... // Do the real work
    } catch (...) {
        // Catch all exceptions
        delete wf; // Handle the deletion, and
        throw; // rethrow the exception: something bad happened
    }
    delete wf; // Question: why not in try block?
}
```

This is good fix, but it is ugly. If there are a lot of allocations, one needs a lot of try-catch blocks.

PMOOP(0396A)-7.11

"Resources acquisition is initialization"

Can we tell the system that `**wf` must be deleted whenever `main` exits in any way, just like a statically allocated variable?

It turns out that this is possible. All we need is to make one more class, and one more variable. The class looks like this:

```
class WavFileAutoClean {
public:
    WavFileAutoClean(WavFile *ptr): wfptr(ptr) {};
    ~WavFileAutoClean() { delete wfptr; }
    WavFile *wfptr;
};
```

That is, we can create a statically allocated variable `wf` of type `WavFileAutoClean` and initialize it using a `WavFile` pointer `ptr`. If the static `wf` is destroyed, the dynamic `ptr` is deleted.

PMOOP(0396A)-7.12

Using the auxillary class

Now we can write the `main` function like this:

```
int main() {
    WavFile *wf = new WavFile("Test.wav");
    WavFileAutoClean wfac(wf);
    ... // Do the real work
    // Nothing else is needed: everything else is automatic.
}
```

Note that since `wfac` is a static variable, it is managed by the system correctly. When `wfac` is killed by the system, so is `wf`.

This technique of using static variables is usually called "resources acquisition is initialization": when some resource is allocated and need deallocation, we initialize a static variable to make sure that it will be deallocated.

PMOOP(0396A)-7.13

Auto-pointers

The problem is that manually creating a new class everytime we deal with a new kind of resources is very troublesome.

Luckily, C++ has a template mechanism that can automatically generate all these, and the technique is generally used enough for the standard library to support that:

```
#include <memory>
...
int main() {
    WavFile *wf = new WavFile("Test.wav");
    auto_ptr<WavFile> wfac(wf);
    ... // Do the real work
}
```

Here `auto_ptr<WavFile>` is equivalent to our `WavFileAutoClean`, except that it support more operations (e.g., you can use it just like `wf`).

PMOOP(0396A)-7.14

Conclusion

- Exception is a very nice feature of the C++ language.
- However, exceptions cause some linear code to behave non-linearly.
- In some cases this lead to problems. A program free of such problems is said to be exception safe.
- One kind of exception safety problems is more serious, since they cause resources leakage.
- Such problems can usually be avoided by using the "resources acquisition is initialization" technique.
- The standard template class `auto_ptr` allows this technique to be used very conveniently.

PMOOP(0396A)-7.15