

# Zoo

Mike Kroutikov

December 13, 2001

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>utils</b>	<b>1</b>
2.1	Buffer . . . . .	1
2.2	Exception objects . . . . .	2
<b>3</b>	<b>utils.threads</b>	<b>4</b>
3.1	Mutex and its Monitor . . . . .	4
3.2	Interrupted state of a Mutex . . . . .	7
3.3	Thread . . . . .	8
3.4	Fifo . . . . .	8
3.5	Clock . . . . .	8
<b>4</b>	<b>io</b>	<b>8</b>
4.1	ByteReaderInterface . . . . .	9
4.2	BufferedReaderInterface . . . . .	9
4.3	ByteReader . . . . .	10
4.4	BufferedReader . . . . .	10
4.5	FileByteReader . . . . .	11
4.6	FileBufferedReader . . . . .	11
4.7	BufferPipe . . . . .	11

<b>5</b>	<b>io.feeder</b>	<b>12</b>
<b>6</b>	<b>system</b>	<b>13</b>
6.1	MultiplexedStreamInterface . . . . .	13
<b>7</b>	<b>video</b>	<b>14</b>
7.1	VideoDecoder . . . . .	14
<b>8</b>	<b>dll</b>	<b>14</b>
8.1	Dll . . . . .	15
8.2	DllManager . . . . .	15

## 1 Introduction

**Zoo** toolkit is a collection of C++ code to read, decode, and play multimedia streams.

## 2 utils

Package `utils` contains generic objects, useful in many high-level packages.

List of objects defined in the `utils`:

`Buffer` — a simple yet powerful raw buffer.

`Exception` — base class for all **Zoo** exceptions.

`BadFormatException` — to be thrown when format error is detected.

`IllegalArgumentException` — to be thrown when argument is invalid.

### 2.1 Buffer

Object that wraps a buffer. The `Buffer` will re-allocate its internal buffer (if necessary) to match the requested size. This internal buffer can not shrink (object uses its internal buffer until the requested size is greater than the allocated one, and re-allocates it to the bigger one when necessary).

Public methods:

`Buffer(int size = 0);` Constructor. Creates a `Buffer` object. Pre-allocates internal buffer of size `size` if passed value of `size` is greater than zero.

`int show(void *data, int len) throw();` Shows the content of the buffer. Returns number of bytes actually written to the data buffer. If `Buffer` does not have any data, returns 0.

`void *getData(int size) throw();` Returns pointer to the internal buffer, that is at least `size` bytes long. After calling this method, the buffer content must be considered to be garbage, and never read (this method is for populating the `Buffer`). When data is copied, the actual size of the data written should be set with `setDataSize()` method.

`void *getData(void) throw();` Returns pointer to the internal buffer. This method is intended for reading `Buffer` content.

`int getDataSize(void) throw();` Returns the number of data bytes stored in the `Buffer`.

`void setDataSize(int size) throw();` Sets the number of data bytes actually put in the `Buffer`.

`static void Buffer::swap(Buffer *one, Buffer *two) throw();` Possibly the most important method. Swaps contents of two buffers. This is the most effective way to pass `Buffer` content from one object to another, without overhead of moving the bytes around (this method does not copy data, but just swaps pointers to the internal buffers).

## 2.2 Exception objects

All exceptions in **Zoo** are passed by the reference (this is more effective than passing by value, and passing by pointer gives the headache of freeing the object memory somewhere in the exception handler).

All exceptions are sub-classes of the `Exception` object. This “basic” exception object has only one method:

`const char *getReason(void) throw();` That returns a string describing the reason for throwing the exception.

Most specific exceptions will sub-class `Exception` in a trivial way.<sup>1</sup>

Example:

```
class MyException : public Exception {
```

---

<sup>1</sup>Actually all exceptions described here are trivial sub-classes. But you may want to create a non-trivial sub-class of `Exception` in order to pass additional information to the catcher.

```

public:
    MyException(void) : Exception("MyException") { } // default constructor
    MyException(const char *fmt, ...) { // printf-style constructor
        va_list va;
        va_start(va, fmt);
        init(fmt, va);
        va_end(va);
    }
};

```

After this, we can use it in the following way:

```

extern void analyze(int x); // no restriction on the set of exceptions
int increment(int x) throw(Exception &) {
    if(x < 0) throw MyException("Bad argument[%d]: must be positive", x);
    analyze(x); // may throw
    return ++x;
}

int main(int ac, char **av) {
    try {
        increment(atol(av[1]));
    }
    catch(MyException &ex) {
        printf("Wrong argument!\n");
    }
    catch(Exception &ex) {
        printf("Unexpected exception: %s\n", ex.getReason());
    }
    return 0;
}

```

### 3 `utils.threads`

Package `utils.threads` defines utilities for multi-threading, including mutex object, thread object, and several thread-safe queues.

`Mutex`, `Monitor` — mutex object (includes condition var), and its accessor object.

`IllegalStateException`, `InterruptedException` — exceptions used in this package.

`Thread` — thread object.

`Fifo` — thread-safe fifo queue.

#### 3.1 `Mutex` and its `Monitor`

`Mutex` object and `Thread` object are main guys to deal with thread creation and synchronization.

`Mutex`: `Mutex(void)` Creates a mutex (that also is capable of synchronizing on a conditional variable).

This object has no public methods. The only way to use it is to use an accessor object `Mutex::Monitor`. This is to facilitate very strict protocol of mutex locking/unlocking in order to avoid “spilling” of a mutex, i.e. leaving a synchronized block of code without unlocking its mutex (due to oversight or an exception throwing).

`Monitor`

`Monitor(Mutex *mutex)` This is the only way to create `Monitor`. Constructor will automatically lock `mutex`. It can be unlocked only by `Monitor` destructor.

`wait(void) throw(InterruptedException &)` Waits on the monitor’s mutex indefinitely (until some other thread calls `notify` or `notifyAll` on the same mutex).

`wait(long millis) throws(InterruptedException &);`

`notify(void);`

`notifyAll(void);`

Note, that there is no reason for nesting two monitors referencing the same mutex. This is allowed syntax, though<sup>2</sup>

---

<sup>2</sup>Since current implementation of `Mutex` uses simple (and fastest) non-recursive locks, such nesting will result in a deadlock, as well as any attempt to lock same `Mutex` by the same thread twice.

## Examples:

```
Mutex mutex;

...// do something unsynchronized here
{
    Monitor monitor(&mutex);
    ... // synchronized section A
}
...// some more unsynchronized code here

{
    Monitor monitor(&mutex);
    ... // synchronized section B
}
...// even more unsynchronized code follows
```

This example illustrates, that `Mutex` is the synchronization *entity*, ensuring that only one thread is within the synchronization section A or B (important: since both sections use same `Mutex` object, it is impossible for one thread to be within section B if another thread entered section A. `Monitor` is a service synchronization object (its constructor locks `mutex`, and destructor unlocks it). Using `Monitor` as a service object ensures, that `mutex` will be used correctly. For example, if some function within code A throws an exception, and this exception is handled within the block, then it will be processed with `mutex` locked. From the other hand, if exception is not handled within the block A, it will leave the synchronized block, and `mutex` will be automatically unlocked.

You can lock more than one resource with this technique. For example, let `mutexA` locks resource A, and `mutexB` locks some other resource B. Then the following block of code will lock resource A, then lock resource B, then do some (useful) work, then unlock resource B, and finally unlock A.

```
Mutex mutexA; // controls resource A
Mutex mutexB; // controls resource B

... // do something unsynchronized
{
    Monitor monitorA(&mutexA);
    // here we successfully obtained lock
```

```

// for the resource A
...
{
    Monitor monitorB(&mutexB);
    // here we got locks for the both A and B
    ...
}
// here lock B is released, but we
// are still keeping lock A
...
}
... // some more unsynchronized code follows

```

In the example above `Monitor` objects ensure correct nesting of the locks (and the correct order of freeing resources). Again, any programmer action that leaves code (e.g. `break`, `continue`, `return`) will not only release the resources locked, but also will do it in the correct order (reverse to the order of resource allocation).

Example below shows a simple implementation of the thread-safe `Stack`:

```

class Stack {
    void **stack;
    long size;
    long top;
    Cond mutex;

public:
    Stack(long sz = STACK_DEFAULT_SIZE)
        : size(sz), top(0) {
        stack = new (void *)[size];
    }
    ~Stack(void) { delete stack; }

    void push(void *object) {
        Monitor monitor(&mutex);
        // if no space, go sleep
    }
}

```

```

        while(top >= size) monitor.wait();
        stack[top++] = object;
        // wake up threads possibly sleeping at pop()
        if(top == 1) monitor.notifyAll();
    }

void *pop(void) {
    Monitor monitor(&mutex);
    // if empty stack, go sleep
    while(top <= 0) monitor.wait();
    void *object = stack[--top];
    // wake up threads possibly sleeping on push()
    if(top == size - 1) monitor.notifyAll();
    return object;
}
};

```

This implementation suspends a thread calling `pop` method if `Stack` object is empty. Symmetrically, a thread calling `push` will be suspended if there is no room in the stack for the new element. Programmer is responsible for waking up threads sleeping on the monitor when condition changes and must be re-evaluated.

### 3.2 Interrupted state of a Mutex

Mutex object can be *interrupted*. Method `setInterrupted` of the `Monitor` class sets (and resets) the interrupted state. If `Mutex` variable is in the interrupted state, then

- 1 All threads sleeping on this condition will be waken up and receive `InterruptedException` exception.
- 2 Any thread that is calling `wait` method for the `Mutex` that is in interrupted state will get `InterruptedException`.

If necessary, you can restore normal functioning of the `Monitor` object by setting interrupted state to `false` with the `setInterrupted` method.

To avoid deadlocks and looping the following rules must be followed:

- 1 Any object, that can infinitely block execution must be interruptable. This is achieved by providing a method that will set states of all `Mutex` variables used

within the object. Normally, a compound object will export public `setInterrupt()` method that will set the state of all `Mutex` variables that it uses.

- 2 For every thread, the body of its run loop should be enclosed in the `try` block catching `InterruptedException`. This will guarantee that on interrupt all threads will exit their run loop and terminate.

These rules provide basis for painless object termination in the multi-threaded environment.

### 3.3 Thread

Object `Thread` is used to create a new thread. User have to sub-class interface `RunnableInterface` where `run` will define the thread functionality, and then create a `Thread` passing superclass of `RunnableInterface` as a parameter. Created `Thread` object does not start an OS thread yet. You have to issue a `Thread::start` method to actually start a thread.

### 3.4 Fifo

This is a Thread-safe implementation of fifo discipline. This is a template, that takes actual object type to be stored as its (sole) parameter.

Constructor takes a single run-time argument, that is a maximum number of elements in the fifo queue (if this number is exceeded, the `put` method will block until some place is freed. If you omit this parameter, `Fifo` object will accept as many object as you supply without blocking (no limit!).

### 3.5 Clock

This is a timing object. It allows one to make sure that specific actions are taken at a specific time intervals. It is being used to delay video frame displaying until the time interval (determined by the fps rate) is up.

## 4 io

Package `io` deals with input/output abstraction used by **Zoo**. Basically, there are two types of streams: a byte-oriented stream, and a buffer-oriented stream.

List of objects defined in `io`.

`ByteReaderInterface` — abstract interface to all byte-oriented input streams.

`BufferedReaderInterface` — interface to all buffer-oriented input streams.

`ByteWriterInterface` — interface to all byte-oriented output streams.

`BufferWriterInterface` — interface to all buffer-oriented output streams.

`BufferedReader` — converts a byte-oriented stream to the buffered one.

`BufferReader` — converts a buffer-oriented stream to the byte-oriented one.

`BitReader` — reads bits from a buffered stream.

`FileByteReader` — file as a raw (byte-oriented) input stream.

`FileBufferReader` — file as a buffered input stream.

`FileByteWriter` — file as a raw output stream.

`FileBufferWriter` — file as a buffered output stream.

`EndOfStreamException` — is thrown to indicate the end of input stream.

`IOException` — problems with IO.

`BufferPipe` — (Circular) pool of buffers that can be used as a (buffered) pipe to pass data from one thread to another.

## 4.1 `ByteReaderInterface`

Interface that defines standard methods to be implemented by any byte-oriented input stream (e.g. `FileByteReader`). The methods are:

- `int read(void *data, int len) throw();` Method to read from stream Returns actual number of bytes read.
- `int readByte(void) throw(EndOfStreamException &);` Method to read byte stream byte-by-byte (may be slow in some implementations). Returns (not negative) number in the range 0–255 inclusively. Throws `EndOfStreamException` on end-of-stream condition.

## 4.2 `BufferReaderInterface`

Interface that defines standard methods to be implemented by any buffer-oriented input stream (e.g. `FileBufferReader`). The methods are:

- `void read(Buffer *buffer) throw(EndOfStreamException &);`  
Populates Buffer `buffer` with some content. Data put into `buffer` must be at least one byte long.

`int show(void *data, int len) throw();` Shows the next `len` bytes in the stream. Returns actual number of bytes showed. The semantics of this method is implementation-dependant. It is perfectly OK to have an implementation, that does nothing in this method and always returns zero (meaning no data available for showing). However, some applications may need the possibility to sneak into stream content without actually reading it. In this case this method should be implemented in a non-trivial way. See the discussion of pre-fetched buffer stream model in `FileBufferStream` for more details on the meaning and usage of `show` method.

### 4.3 ByteReader

An object, that converts a buffer-oriented stream into the byte-oriented one. Implements `ByteReaderInterface`.

Constructor: `ByteReader(BufferReaderInterface *reader) throw();`

Though `BufferReaderInterface` is faster in passing data from one object to another, it lacks methods that are necessary to parse stream content. Parser might want to convert incoming `BufferReaderInterface` into `ByteReaderInterface` before actually parsing the data. This convenience object will come handy in that case.

### 4.4 BufferReader

An template, that converts a byte-oriented stream into the buffer-oriented one. Implements `BufferReaderInterface`. The single template parameter is an integer—the blocking factor (showing what should be the size of the emitted buffers).

Constructor: `BufferReader<BUFFER_SIZE>(ByteReaderInterface *reader) throw();`

Normally, it is preferrable to deal with a buffer-oriented streams (no data-copying overhead when passing data from one object to another). You can use this object to perform a conversion from `ByteReaderInterface` to `ByfferReaderInterface`.

The implementation adopts pre-fetching paradigm, and implements non-empty `show` method. Let me discuss this in some more detail. There are two possible approaches:

- read-on-demand. In this approach, the input byte stream is being read only when a new buffer of data is being requested from `BufferReader`. In this case, the data from the input stream is being stored in the buffer and passed to the caller.
- pre-fetching. In this approach, `BufferReader` reads one buffer ahead. When

a new buffer of data is being requested, the pre-fetched buffer is returned to the caller, while `BufferedReader` reads next one in.

The advantage of the pre-fetching approach is that we can sneak into the pre-fetched buffer content to determine what data is coming next. The disadvantage is that at the time of the object creation, the `reader` object (passed as a parameter to the constructor) must be ready to be read from.

## 4.5 FileByteReader

Byte-oriented input stream originated from a file.

Implements `ByteReaderInterface`

Methods:

`FileByteReader(const char *path) throw(IOException &);` Constructor. Argument `path` gives a full path to the file to be opened.

## 4.6 FileBufferReader

Buffer-oriented input stream originated from a file. This is a template, taking one integer parameter - the blocking factor. For example:

```
FileBufferReader<1024> reader("./data.dat");
```

Creates an input buffer-oriented stream. The Buffers emitted by this stream will be 1024-bytes long.

Constructor: `FileBufferReader<BUFFER_SIZE>(const char *path) throw(IOException &);` Creates a stream that reads file `path`.

The implementation of this object is quite straightforward—we take `FileByteReader` (that is the native representation of file API), and convert it to byte stream with `BufferedReader` template.

## 4.7 BufferPipe

This is an unspillable pool of `Buffers` very convenient to pass buffer stream from one thread to another (it is also thread-safe!). Unspillable means, that you can not possibly lose a buffer from this pool. It contains two `Fifo` queues — one for the buffers with data (pushed buffers), other — for the blank buffers (i.e. buffers used by the receiver). This object is designed to be used as a communication pipe (buffer-oriented pipe) between two different threads. Example: thread A writes buffers to the pipe, while thread B reads buffers from the pipe.

Implements `BufferWriterInterface`, `BufferReaderInterface`.

Constructor: `BufferPipe(int allocSize, int bufferSize);`

This object is implemented as a pair of `Fifo` streams. One stream holds buffers written to the pipe. Naturally, “reader” takes buffers from this queue, while “writer” puts data there. Since read/write is actually implemented as buffer swap operation (for efficiency), we need the second `Fifo` stream to hold used buffers. Constructor parameter `allocSize` defines how many buffers pipe has. Second parameter `bufferSize` indicates the size of the initially allocated buffers in the pipe. Note, that during read/write process, buffer content is being swapped with reader/writer data, and, therefore, `bufferSize` is transient.

Methods:

- `void read(Buffer *other) throw(IOException &);` Method to read from the pipe. If no data was written, this method will block.
- `bool tryRead(Buffer *other) throw(IOException &);` Same as `read`, except it never blocks (returns `false` instead).
- `void write(Buffer *other) throw(IOException &);` Method to write into the pipe. If no space in the output `Fifo`, will block.
- `bool tryWrite(Buffer *other) throw(IOException &);` Same as `write`, except it never blocks (returns `false` instead).
- `int show(void *buffer, int len) throw ();` Shows the content of the next buffer to the reader.
- `void setInterrupted(bool what);` Sets/resets the *interrupted* state of the object. As usual, if `BufferPipe` is in the interrupted state, any attempt to read/write will generate exception. This method is useful for bringing multi-threaded applications in a consistent state (e.g. breaking deadlocks).

## 5 io.feeder

This package defines just one object — a `Feeder`, that converts a buffered stream into other buffered stream (possibly changing its blocking factor). This object can be used, for instance, for changing buffering parameters. Originally, this object was created to cope with the (weird — IMO) API for audio decoders in Microsoft dlls.

## 6 system

### 6.1 MultiplexedStreamInterface

Demultiplexor is an object, that takes a buffer stream (that supposedly contains multiple elementary streams multiplexed) and allows to extract all or any subset of the elementary streams. It provides a directory look-up service (directory of streams available for de-multiplexing). Then, user can decide which streams are to be de-multiplexed, and get a `BufferReaderInterface` object for every such a stream. Now it is ready to read from elementary streams.

If caller wants to de-multiplex more than one stream (that is usually the case, when we read one video and one audio streams), he must use multi-threading in order to read from each stream independantly. This is because caller does not know (and does not have any means to know) in which particular order elementary packets are stored in the multiplexed stream, therefore it does not know in which sequence he must request buffers for elementary streams. Multithreaded implmentation must ensure that every elementary stream is being read by different thread. Then, if a particular elementary stream does not have data ready yet (e.g. it is a long way down the multiplexed stream), the `read` method for this elementary stream will block the thread until the data is available.

A generic properties of demultiplexor are described by its interface `MultiplexedStreamInterface`:

```
void open(void) throw(BadFormatException &); Opens (reads) multiplexed stream directory. Must be called only once, just after creating demultiplexor object. After open (if it succeeds, of course) one can list which elementary streams are available for de-multiplexing.
```

```
int getFirstVideoStreamID(void) throw(); Returns non-negative integer, that is an id of the first multiplexed stream. If there are no video streams available, returns -1.
```

```
int getNextVideoStreamID(int prevID) throw(); Returns non-negative integer, that is an id of the next video stream. If no more video streams, returns .
```

```
BufferReaderInterface *getBufferReader(int id) throw(BadArgumentException &); Informs demultiplexor that video stream with the id id is active (non-active elementary streams are just skipped in the process of de-multiplexing), and returns a BufferReaderInterface object to be used to read from this elementary stream. Caller must read from this stream (or other streams may get dead-locked).
```

The sequence of applying these methods is *important*. Namely, following rules should apply:

- `open` should be called immediately after constructing a `MultiplexedStreamInterface` object. You can not apply any other methods before `object`.
- `open` and directory services are *not* MT-safe. Therefore, multithreaded part of a program should start after `getBufferedReader`.

## 7 video

### 7.1 VideoDecoder

`VideoDecoderInterface` describes common properties of all video decoders. A video decoder takes a buffer stream as its input, and produces set of video frames suitable for displaying on the screen.

`setPreferredFormat(int fmt) throws()` gives a hint to the decoder which output format is preferable by the receiver. There is no guarantee, that this request will be granted. In fact, MPEG decoder implementation just ignores this hint. However, AVI decoders (were typically developer does not have direct access to internal YUV buffer) may grant it thus improving decoding performance by eliminating a need for an adapter. This method must be called *before* decoder is opened.

`void open(void) throw(BadFormatException &);` Opens decoder. Must be called just after creating the object, and before requesting the first frame.

`int getWidth(void) throw()` returns width of the decoded frame (method belongs to the `PixBufferInterface`). This method should not be called prior to the `open`.

`int getWidth(void) throw()` returns width of the decoded frame (method belongs to the `PixBufferInterface`). Result is valid only if decoder object was opened.

## 8 dll

This package contains abstractions for loading libraries dynamically. This technology is used by **Zoo** plugins (however, the objects themselves are generic enough to be used by a third-party application).

A dynamic library in **Zoo** format must export following global symbols

- `unsigned long dllSignature` — this a signature, that tells that library follows conventions described here. This symbol allows user to detect (and discard) libraries that do not belong to **Zoo** package. The value should be set to `Dll::SIGNATURE`.

- `double dllVersion` — this is a format version. Current format version is 0.1.
- `void *dllService(int svc)` — this is the main entry point, used to get factory functions for the different services. From the format point of view, this entry just must exist. Its usage is described below.

A dll exports one or more *services*. A service is identified by an integer id. Value 0 is prohibited. Values from 1 to 1024 inclusively are “well-known” services. This means, that these services (if defined) must be listed in `DllRepository`. This prevents id clash. All other id values can be used freely. However, if you develop a plugin that may be interesting to anyone except you, please, pick a “well-known” service id and resiter this service in `DllRepository`.

When dll entry point is called, it is supposed to return either `null` (if dll does not provide this service), or something, that is not `null` and can be used to start the service. What exactly is being returned depends on the service type. Normally, this will be a factory function, capable of generating service instances (see example in `examples/dll` for a complete working example).

## 8.1 Dll

`Dll` represents a dynamic library that follows the conventions described above. It takes file name of a library as its constructor argument. Constructor will load the library. If format is wrong, a `BadFormatException` will be thrown.

`getService(int id)` — tries to get a service factory from the dll for service id. Returns `null` if no such service defined in the dll.

## 8.2 DllManager

Searches a list of directories for plugins, tries to load every one and find a service requested. Constructor takes a list of directories to be searched. Note, that only files containing `*io*` pattern are considered to be loadable.