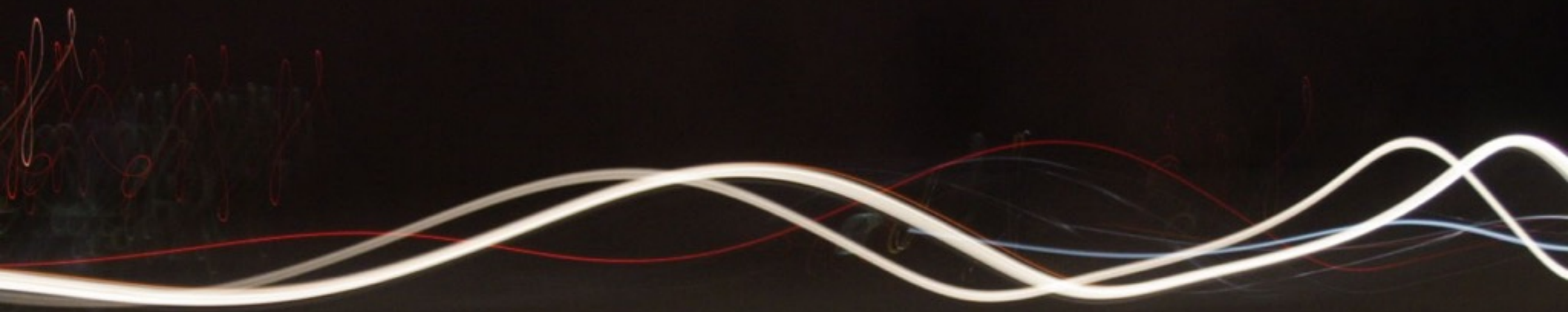
C++ in the audio industry



Timur Doumler

CppCon, 21 September 2015

real-time programming

fast & efficient DSP

lock-free thread synchronisation

embedded systems in music hardware

SIMD and memory alignment

cross-platform challenges

real-time programming

fast & efficient DSP

lock-free thread synchronisation

embedded systems in music hardware

SIMD and memory alignment

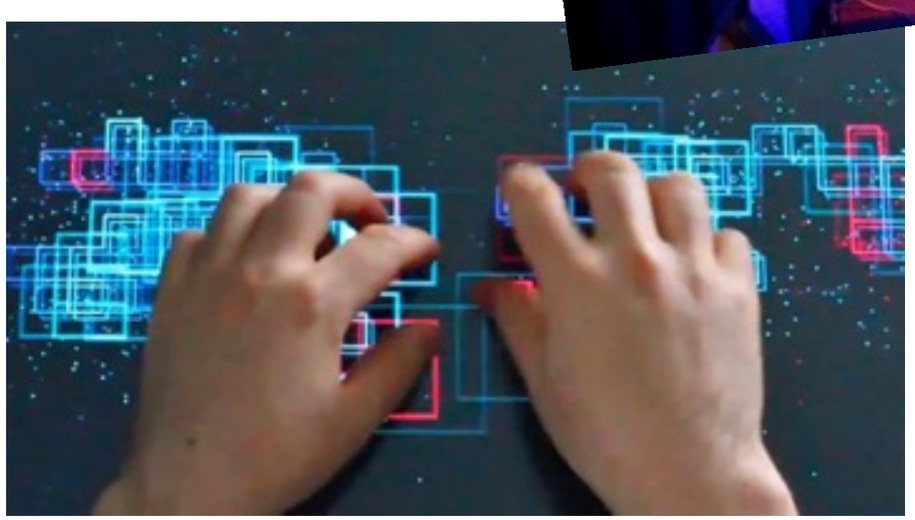
cross-platform challenges

- introduction
- representing audio in C++
- audio = “hard real-time” programming
    - consequences
    - dos and don’ts
- lock-free thread synchronisation
    - sharing objects between threads
    - exchanging data between threads
    - memory management / object lifetime

music producers
audio engineers
composers
sound designers

audio apps
games
multimedia

live performers
djs
touring musicians

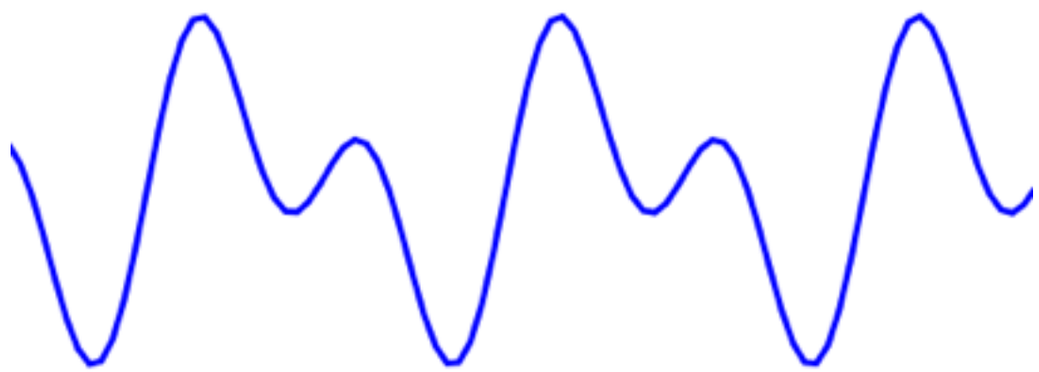


science, art, creative coding

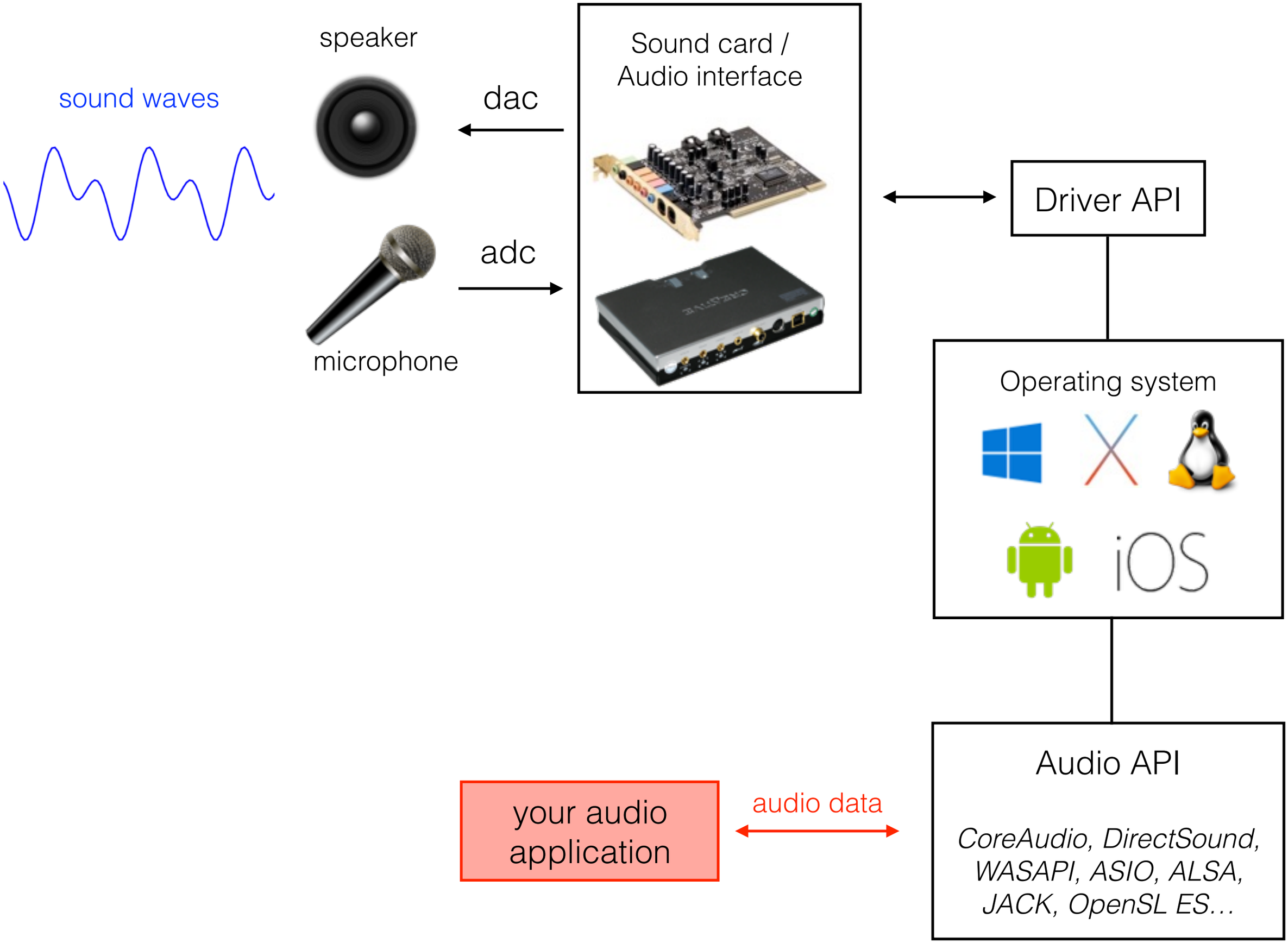
developing musical instruments,
synthesisers, effects

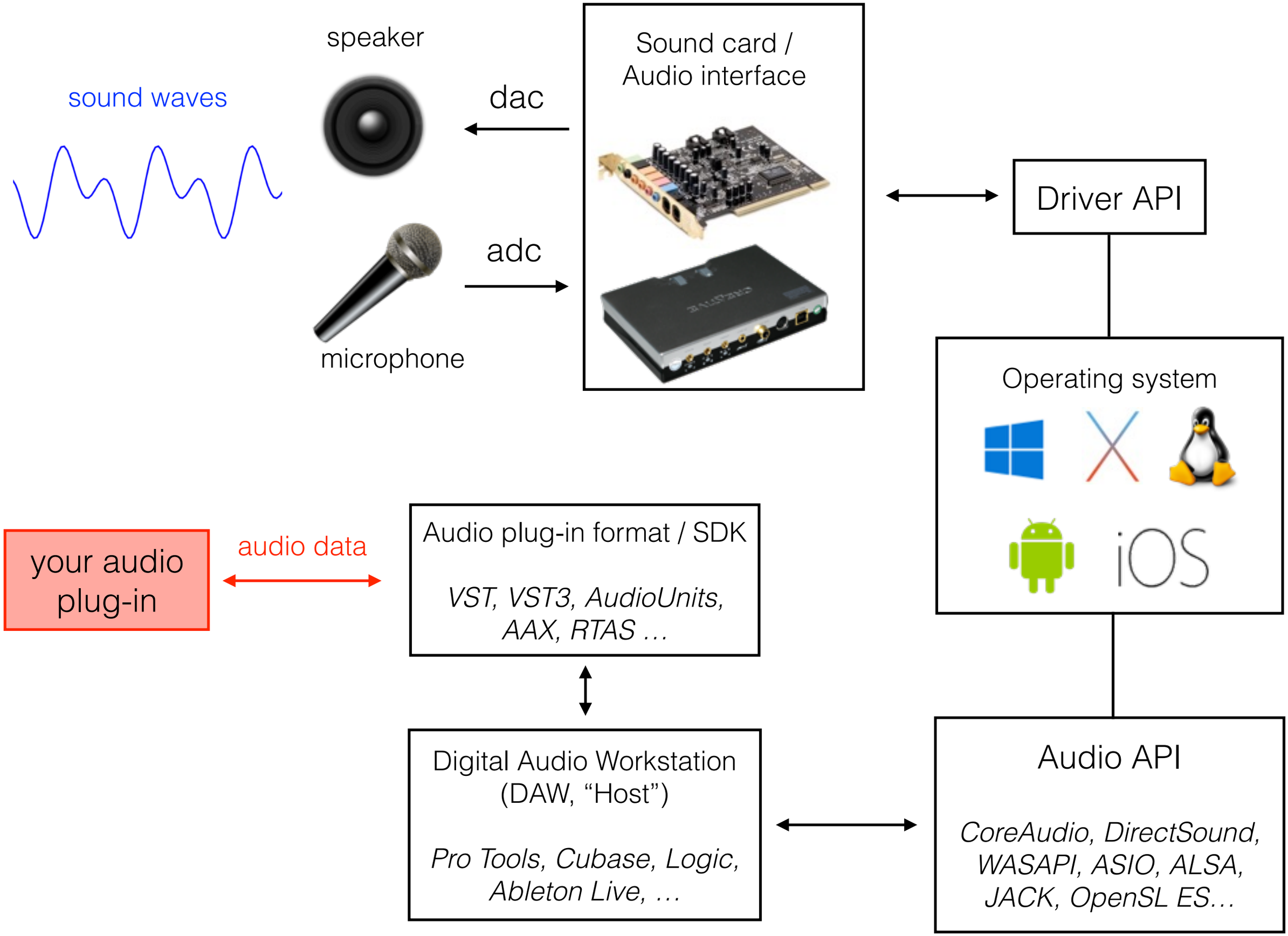

bright
volume
bass
middle

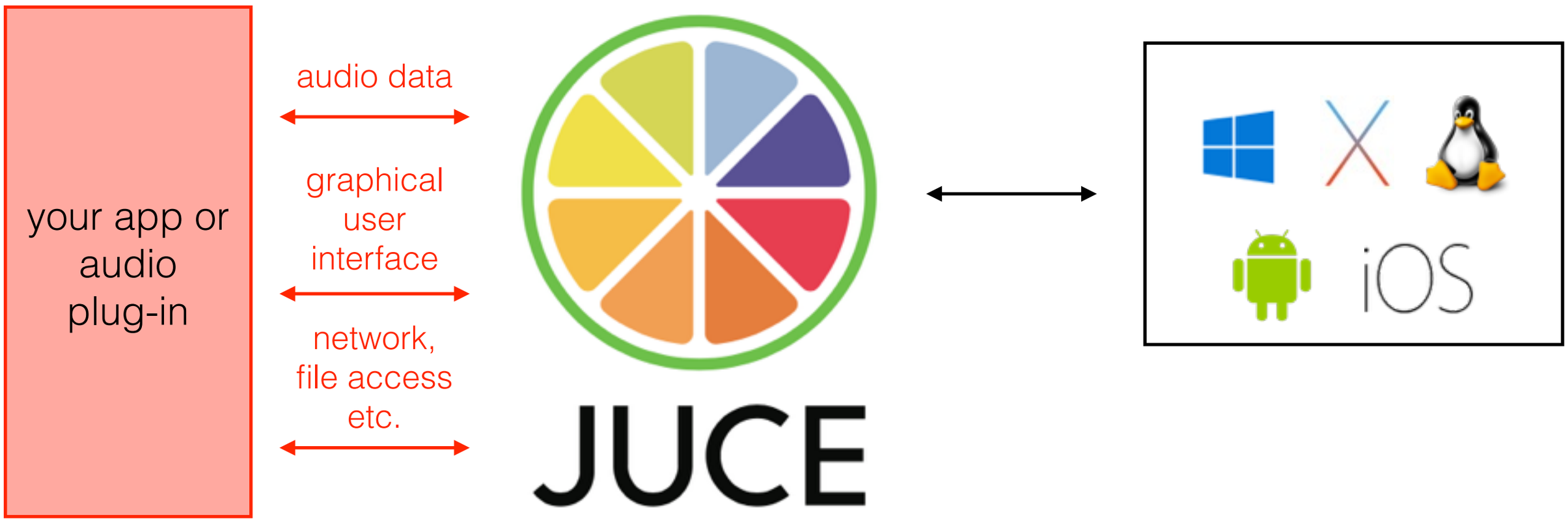
audio = data representation of sound waves

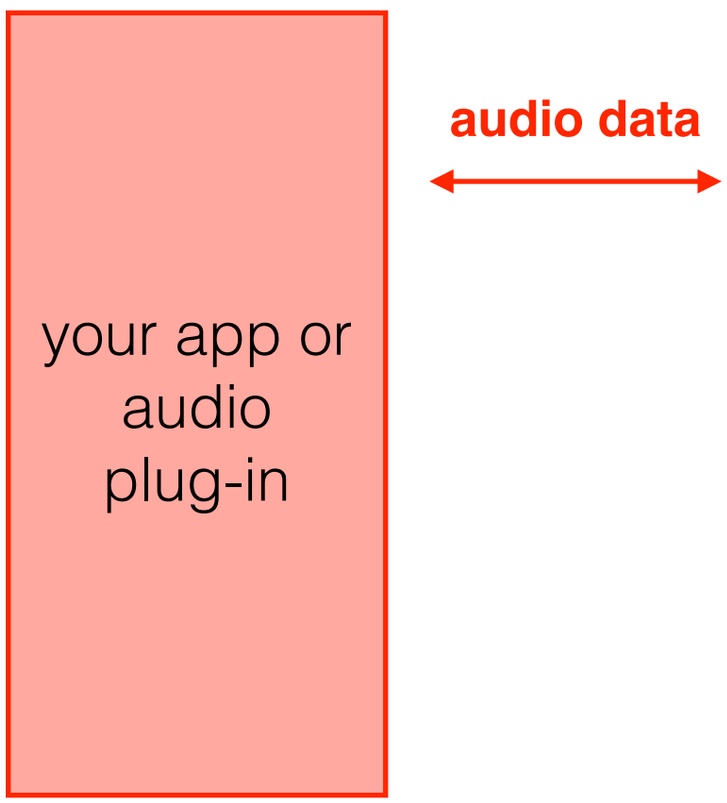


- Audio data can be represented and manipulated digitally
- The C++ language has no concept of audio
- But there are conventions, 3rd-party libraries, APIs

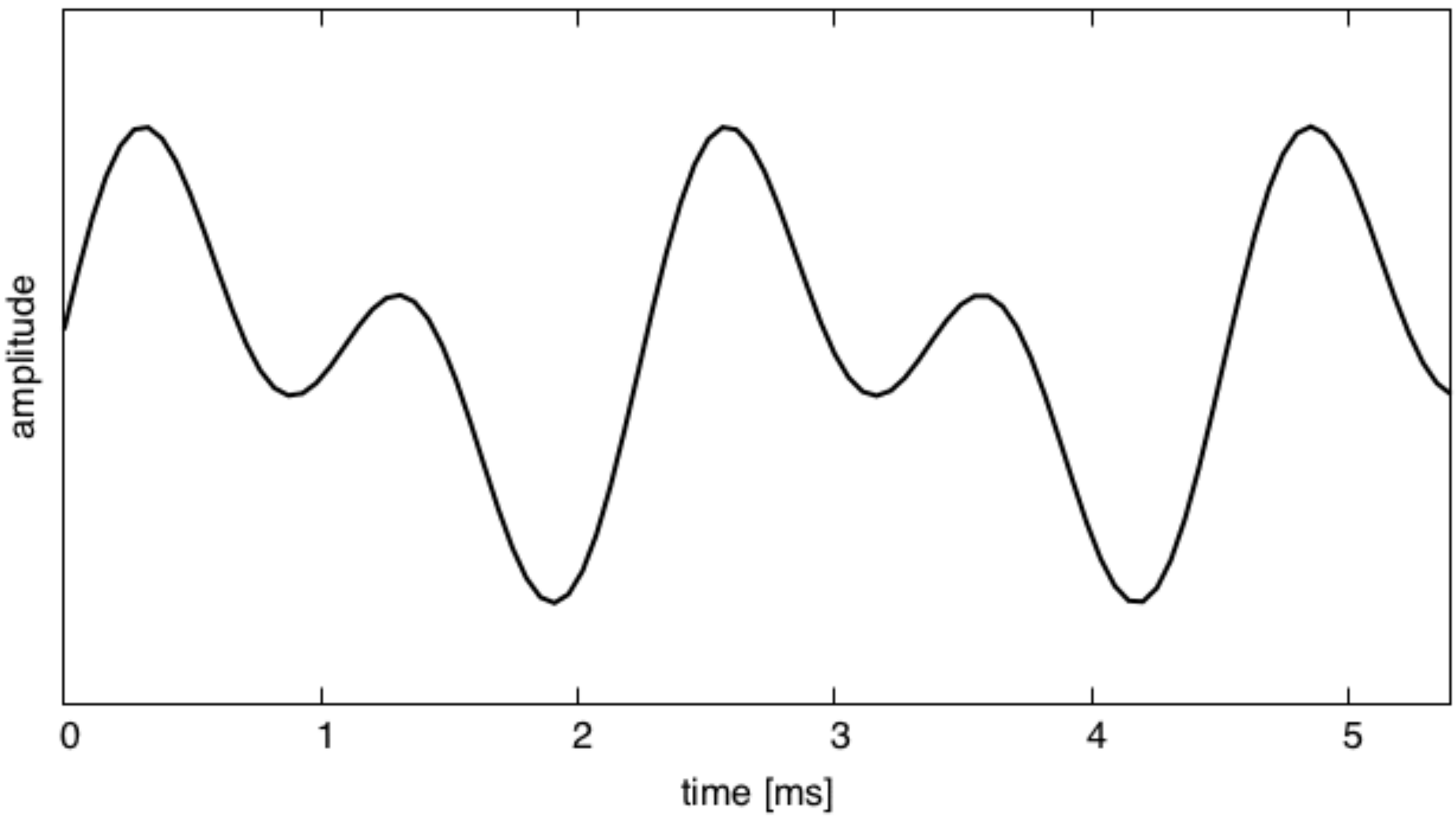

sound waves
speaker
dac
Sound card /
Audio interface
adc
microphone
Driver API
Operating system
iOS
Audio API
CoreAudio, DirectSound,
WASAPI, ASIO, ALSA,
JACK, OpenSL ES…
your audio
application
audio data


sound waves
speaker
dac
Sound card /
Audio interface
Driver API
adc
microphone
Operating system
iOS
your audio
plug-in
audio data
Audio plug-in format / SDK
VST, VST3, AudioUnits,
AAX, RTAS …
Digital Audio Workstation
(DAW, “Host”)
Pro Tools, Cubase, Logic,
Ableton Live, …
Audio API
CoreAudio, DirectSound,
WASAPI, ASIO, ALSA,
JACK, OpenSL ES…


your app or audio plug-in
audio data
graphical user interface
network, file access etc.
JUCE
iOS


audio data
your app or audio plug-in

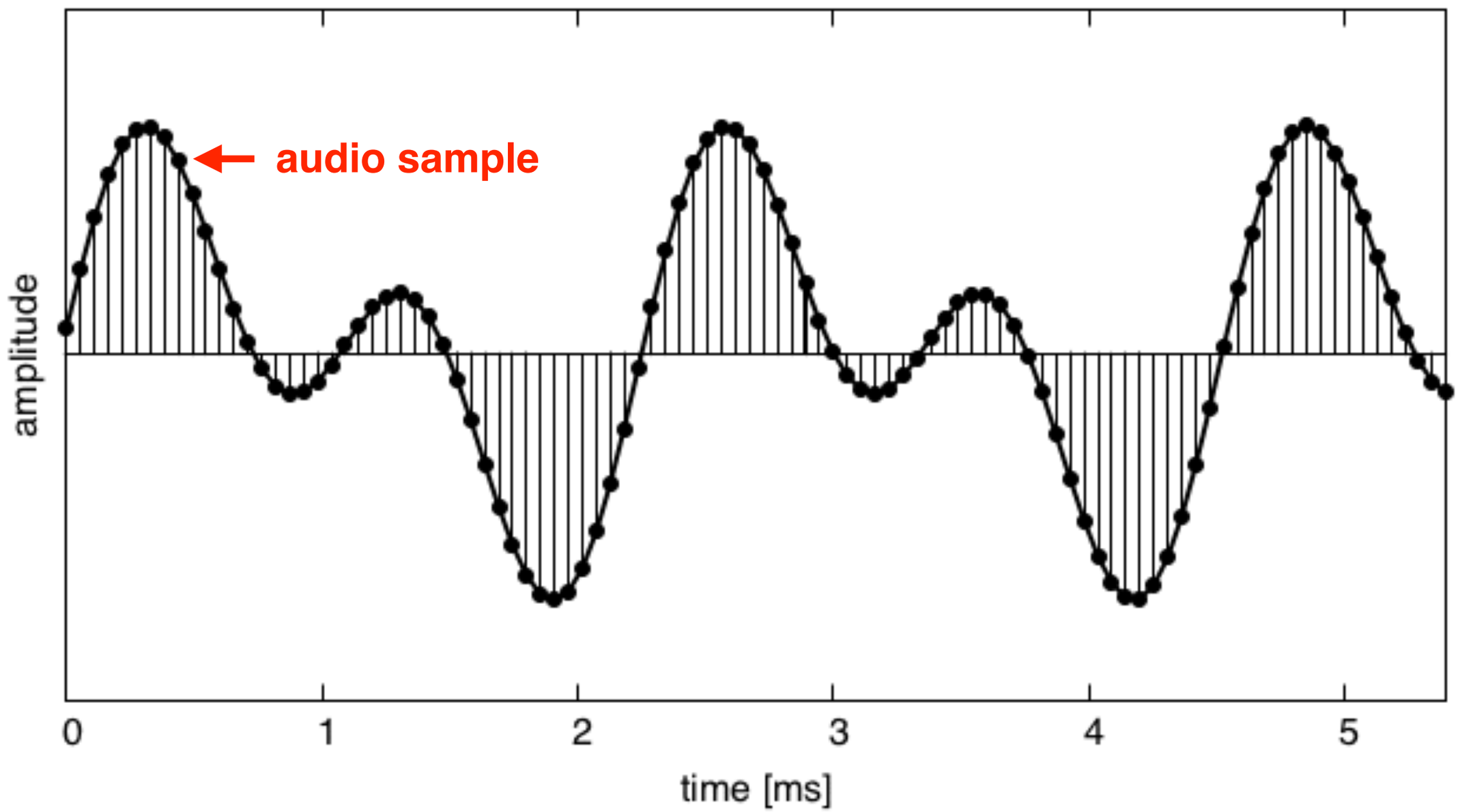
Representation of audio data in C++



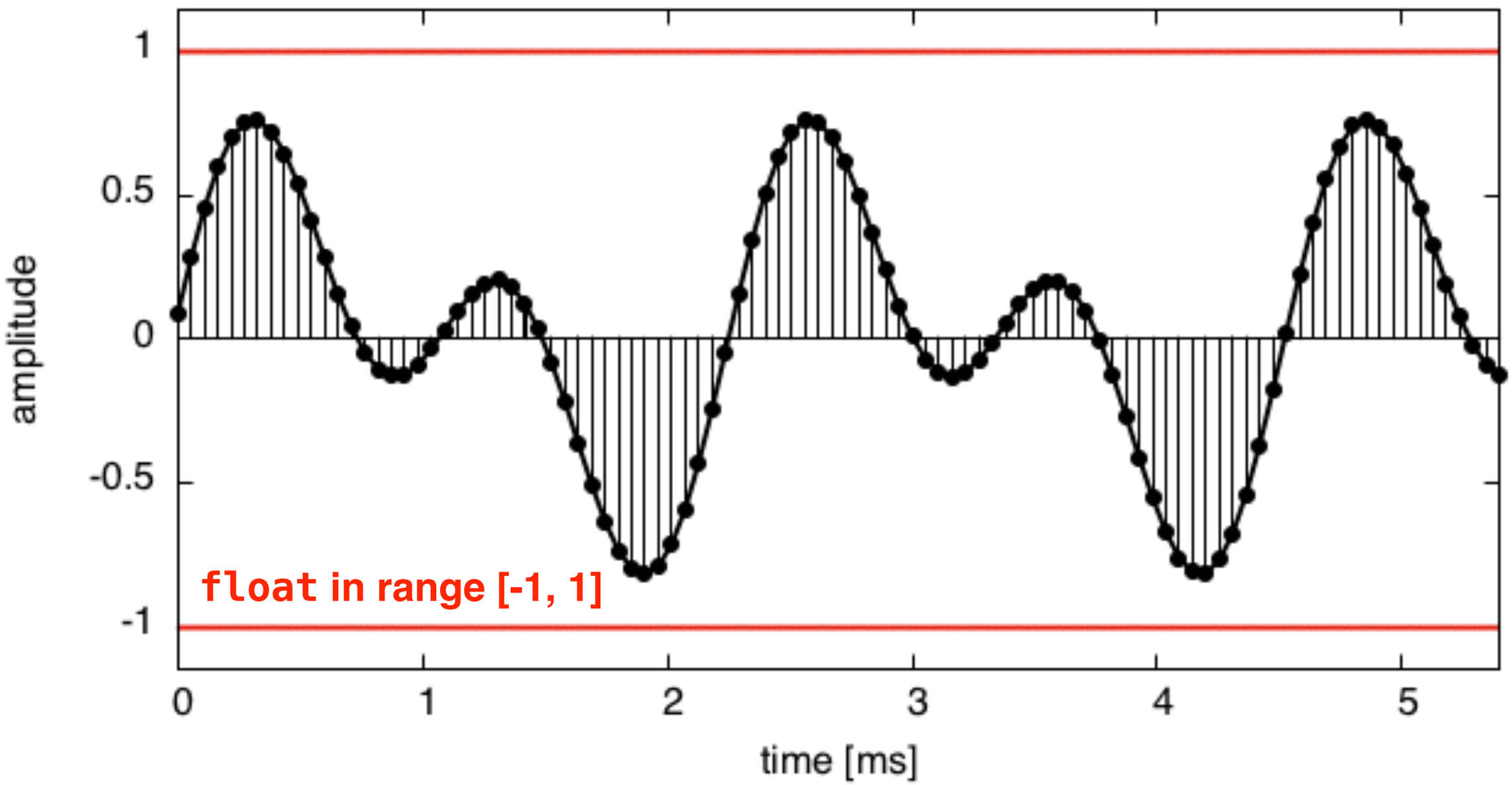
Representation of audio data in C++

sample rate

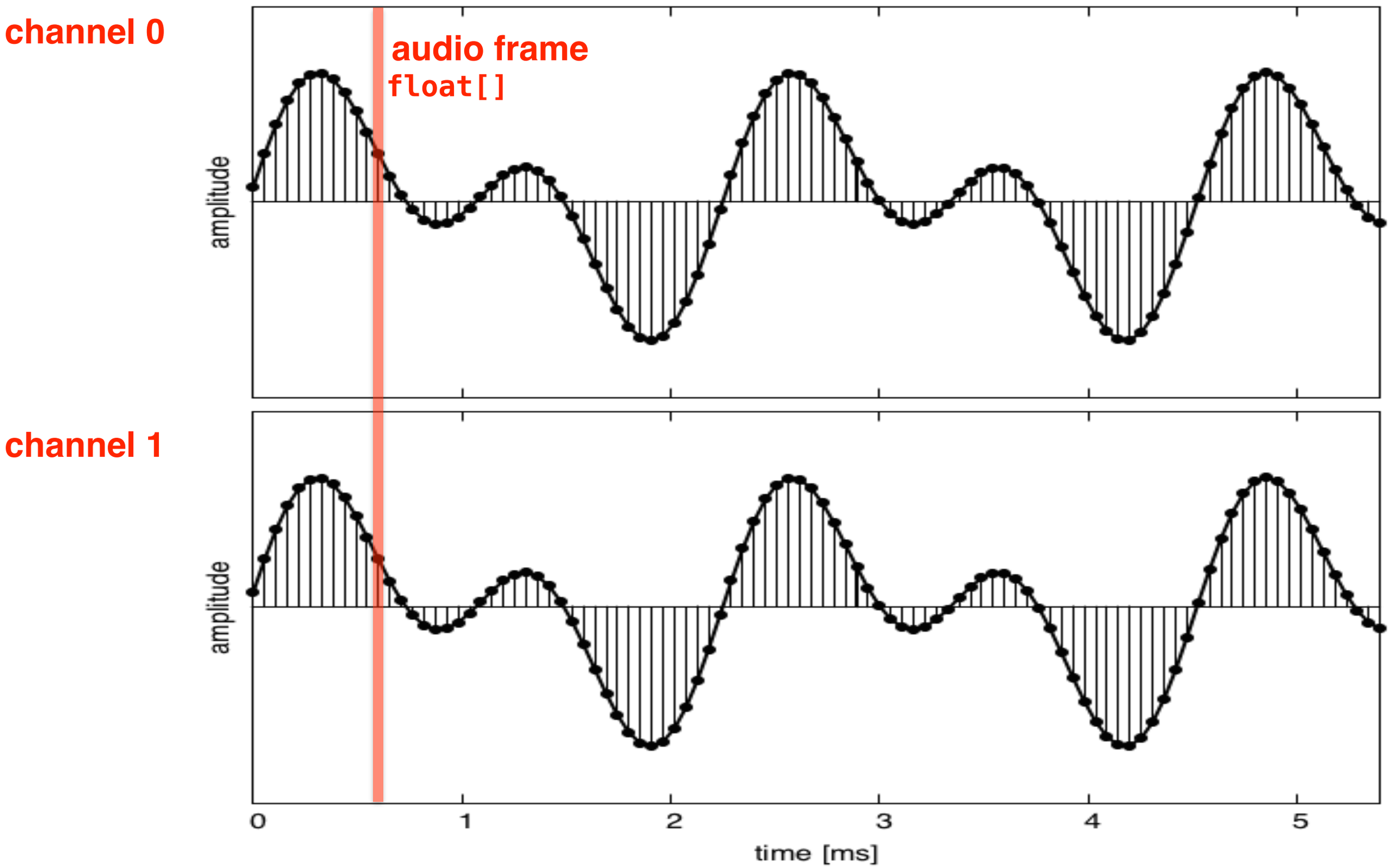
44.1 kHz, 48 kHz, 96 kHz, …



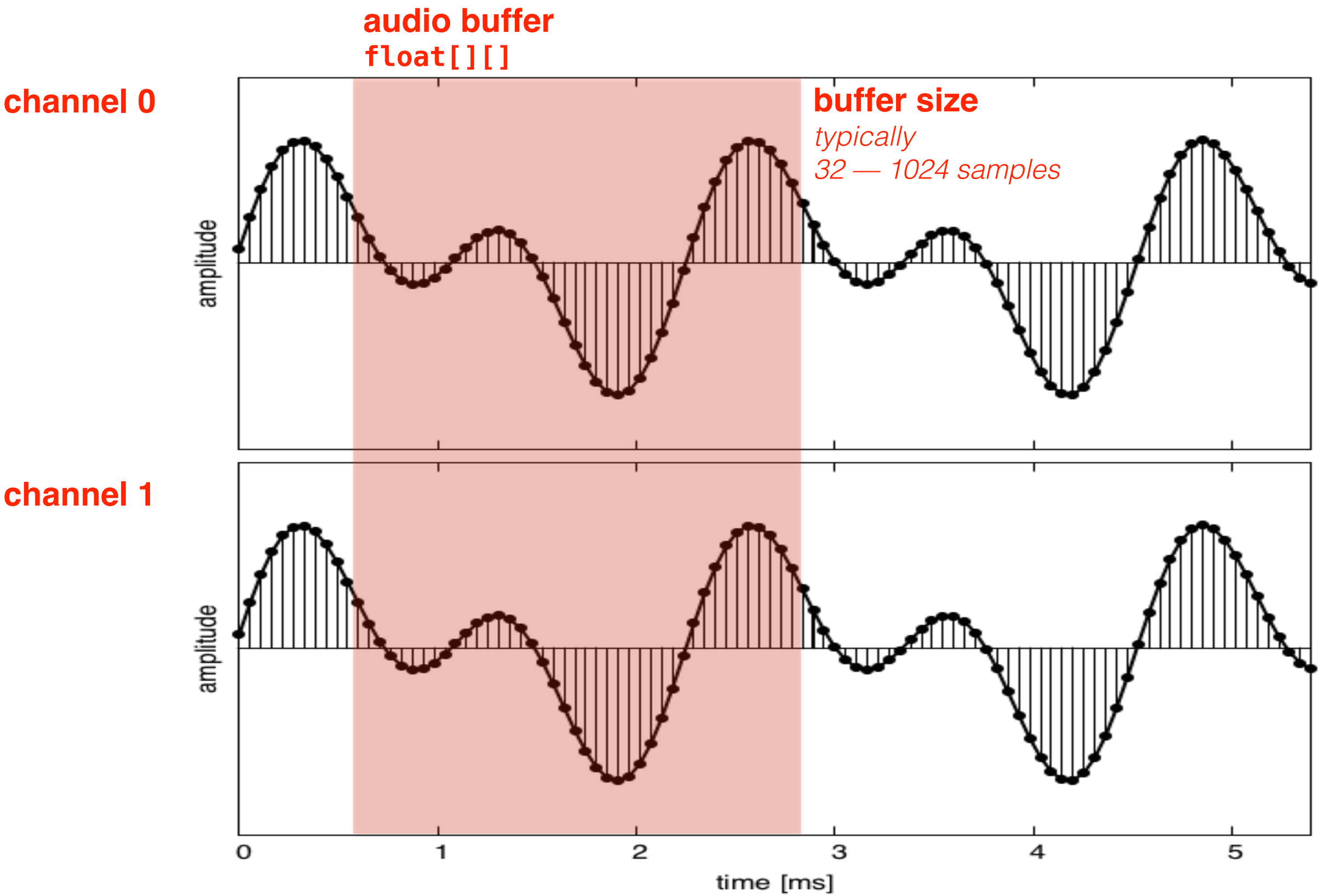
Representation of audio data in C++



Representation of audio data in C++



Representation of audio data in C++



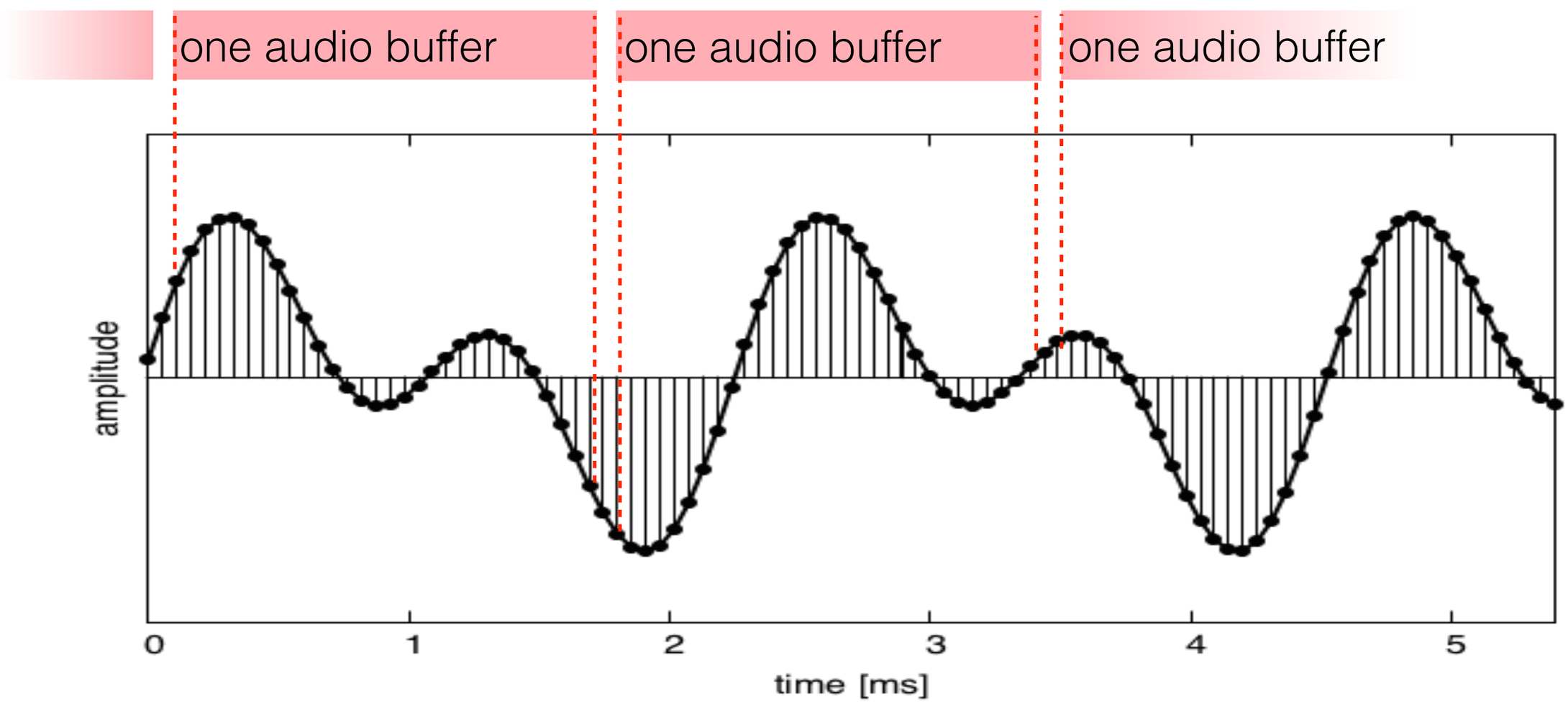
The audio callback

```
void audioCallback (float** channelData,
                    int numChannels,
                    int numSamples)
{
    for (int channel = 0; channel < numChannels; ++channel)
        for (int sample = 0; sample < numSamples; ++sample)
            channelData[channel][sample] = 0.0f;
}
```

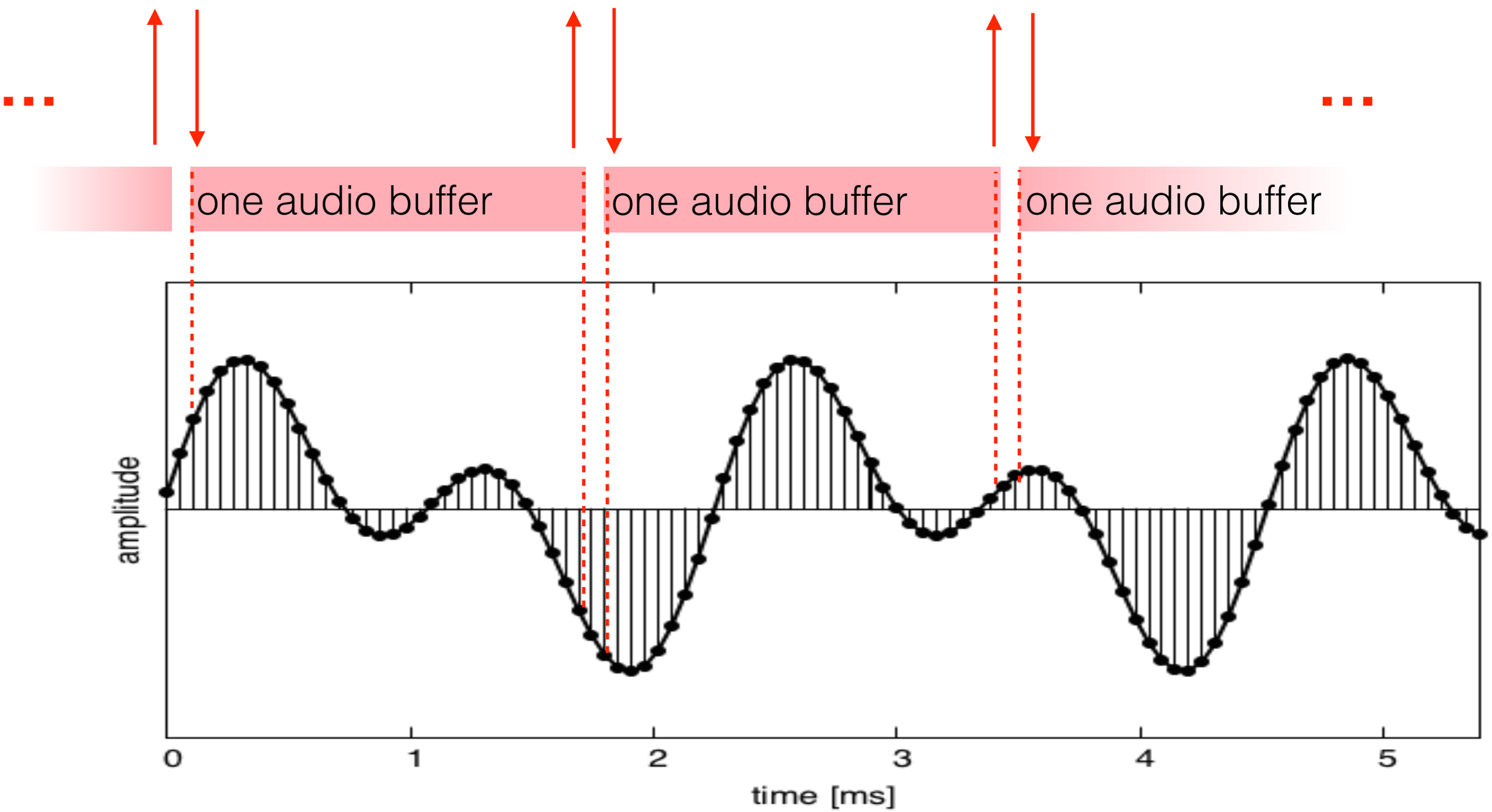
The audio callback

```
void audioCallback (float** inputChannelData,
                    int numInputChannels,
                    float** outputChannelData,
                    int numOutputChannels,
                    int numSamples)
{
    // ...
}
```

audio programming is
“hard real-time” programming


one audio buffer
one audio buffer
one audio buffer
amplitude
0
1
2
3
4
5
time [ms]

hard real-time audio callback



#0 rule of audio code:

the audio callback
waits for <u>nothing</u>.

Buffer size	**Buffer length** @ sample rate 44.1 kHz	**Buffer length** @ sample rate 96 kHz
32 samples	0.73 ms	0.33 ms
64 samples	1.45 ms	0.66 ms
128 samples	2.90 ms	1.33 ms
…	…	…
1024 samples	23.2 ms	10.7 ms

<u>Total system audio latency:</u>

< 10 ms: *good for interactive audio performance*

10-30 ms: *noticeable*

> 30 ms: *OK if response doesn't have to feel real-time*

```
void audioCallback (float** channelData,
                    int numChannels,
                    int numSamples)
{
    for (int channel = 0; channel < numChannels; ++channel)
        for (int sample = 0; sample < numSamples; ++sample)
            channelData[channel][sample] = ...;
}
```

…complex DSP calculations here…

we have to <u>guarantee</u> that:

- this function will return in time < buffer length
- will finish processing the buffer
- the output buffer will contain <u>valid audio data</u> afterwards
- there will be no error / exception / …

```
void audioCallback (float** channelData,
                    int numChannels,
                    int numSamples) noexcept
{
    for (int channel = 0; channel < numChannels; ++channel)
        for (int sample = 0; sample < numSamples; ++sample)
            channelData[channel][sample] = ...;
}
```

…complex DSP calculations here…

we have to <u>guarantee</u> that:

- this function will return in time < buffer length
- will finish processing the buffer
- the output buffer will contain <u>valid audio data</u> afterwards
- there will be no error / exception / …

```
void audioCallback (float** channelData,
                    int numChannels,
                    int numSamples) noexcept
{
    for (int channel = 0; channel < numChannels; ++channel)
        for (int sample = 0; sample < numSamples; ++sample)
            channelData[channel][sample] = ...;
}
```

…complex DSP calculations here…

we have to <u>guarantee</u> that:

- this function will return in time < buffer length
- will finish processing the buffer
- the output buffer will contain <u>valid audio data</u> afterwards
- there will be no error / exception / …

otherwise we will cause an **audio dropout**.

#1 rule of audio code:

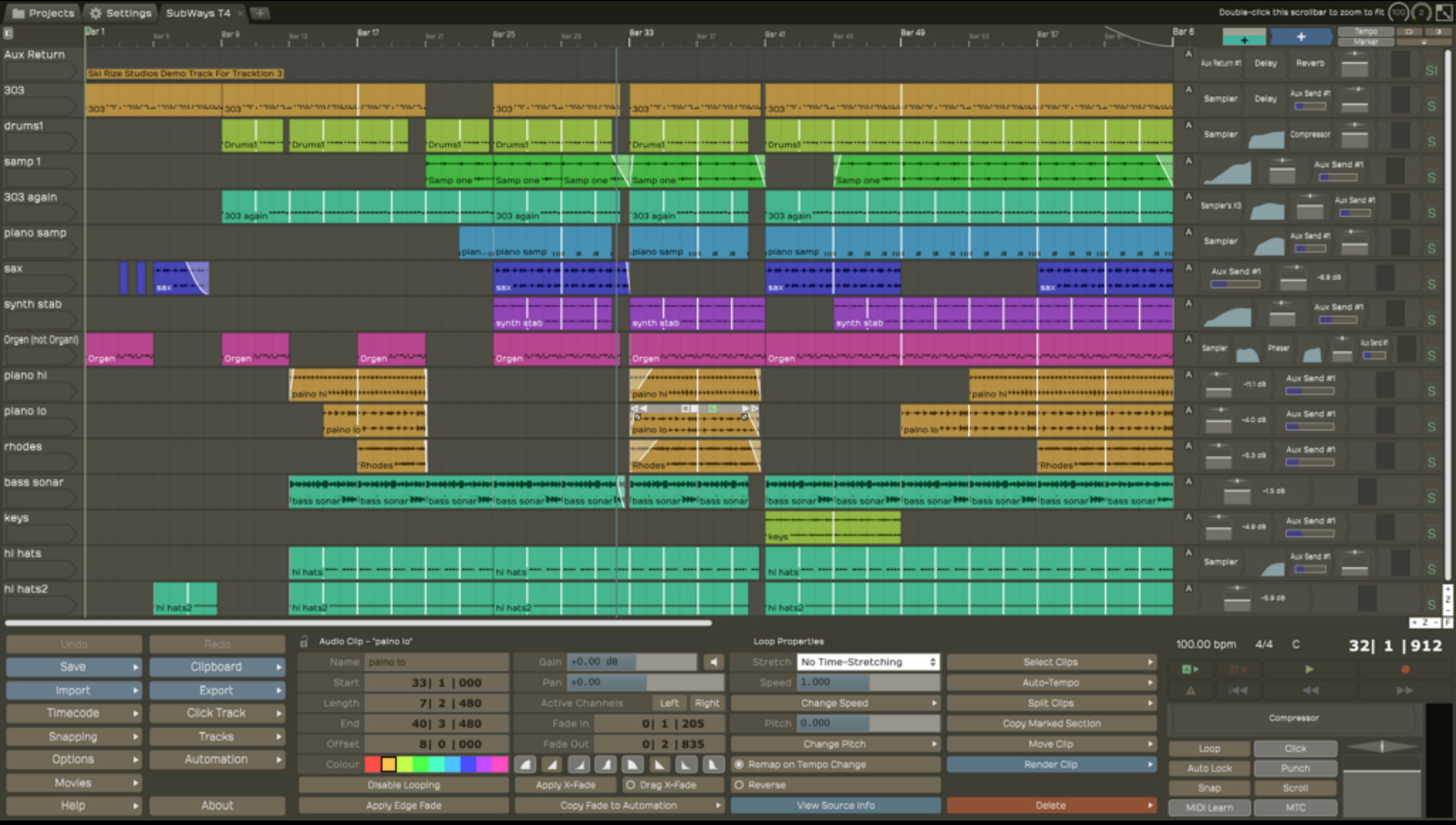
<u>never</u> cause audio dropouts.

#1 rule of audio code:

<u>never</u> cause audio dropouts.

#1 rule of audio code:

<u>never</u> cause audio dropouts.



#1 rule of audio code:

<u>never</u> cause audio dropouts.

- C++ in itself is not a real-time safe language.
- You have to fulfil the real-time guarantees by how you write code.
- There are certain dos and don’ts.

#2 rule of audio code:
don’t do anything
where you can’t predict
how much time it will take.

Don’t block on the audio thread.

- Don’t call anything that blocks / waits
  - `std::mutex::lock()`
  - `while (! std::mutex::try_lock()) { ... }`
  - `std::thread::join()`
- Don’t call anything that <u>blocks / waits internally</u>

Do:

- lock-free programming
  —> *examples follow…*

Why you shouldn’t block / wait in the audio thread.

- reason #1: **you don’t know how long it takes**
  - in most cases you are waiting for code that is not real-time safe
    —> *you get audio drop-outs*

- reason #2: **priority inversion**
  - audio thread is a high-prority system thread
  - when blocked, it is waiting for a lower-priority thread
  - the lower-priority thread may be interrupted by other threads
    —> *you get audio drop-outs*

#3 rule of audio code:

in the audio callback, only run lock-free code.

Don’t (de)allocate any memory on the audio thread.

- don’t call `new` or `delete`
- don’t call anything that uses them internally:
  - constructing RAII objects or let them go out of scope
  - many, many other functions

  —> for example, `std::vector::push_back()` may reallocate!

Do:

- put your data on the stack
- preallocate your data
- be careful with object lifetime and ownership
- use containers from Boost.Intrusive and Boost.Lockfree
- use custom real-time safe containers (lock-free queues, stacks, lists...)
- *for larger systems:* manage your own preallocated, deterministic, lock-free heap

Don’t do any I/O on the audio thread.

- don’t call `std::cout`, `printf` etc.
- don’t do any interprocess communication
- don’t access files on the disk, the network, …
- don’t do any graphics work

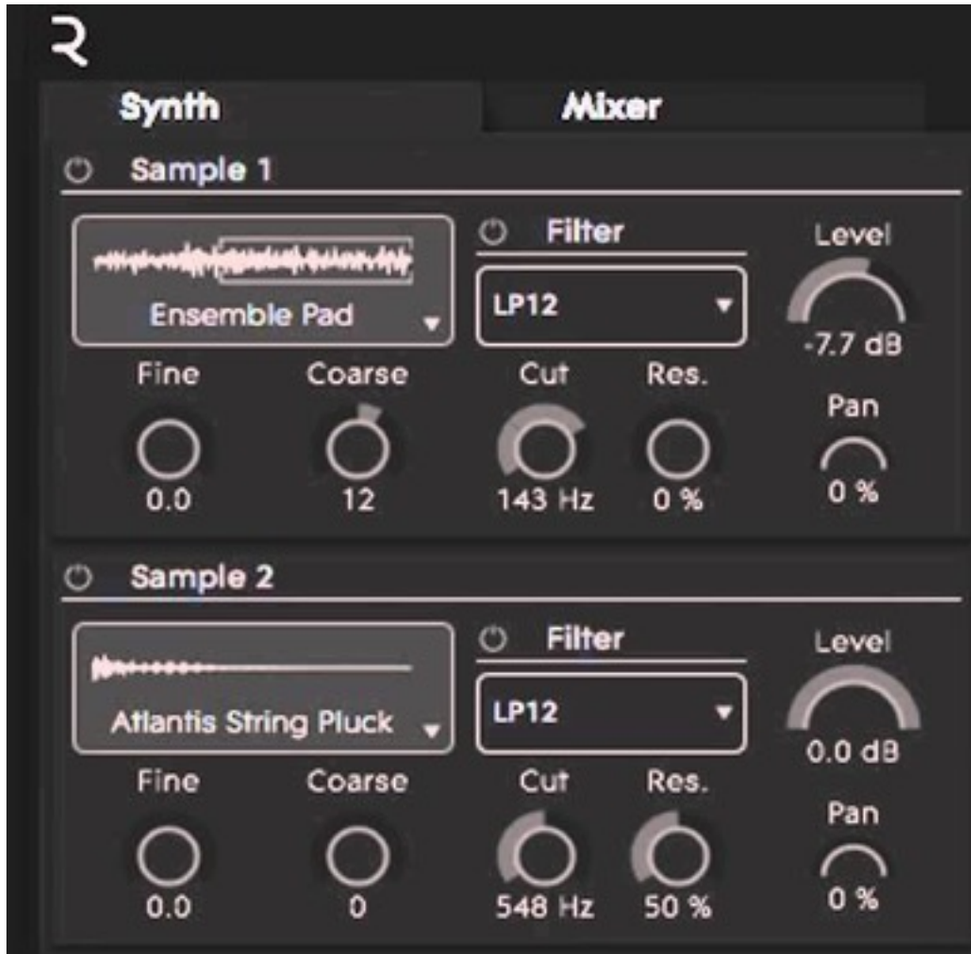
Do:

- use other threads for all of the above
- synchronise them lock-free

- Don’t call any 3rd party code.
  (if you don’t know what’s going on inside and how long it will take)

- Don't call algorithms with unsafe worst-case behaviour.
  basically: O(1) is fine, amortised O(1) is not
  —> for example, `std::unordered_set::insert()` may rehash

- Don’t run into page faults.


Synth
Mixer
Sample 1
Ensemble Pad
Filter
LP12
Level
-7.7 dB
Fine
Coarse
Cut
Res.
Pan
0.0
12
143 Hz
0 %
0 %
Sample 2
Atlantis String Pluck
Filter
LP12
Level
0.0 dB
Fine
Coarse
Cut
Res.
Pan
0.0
0
548 Hz
50 %
0 %


Session Horns
Full Section
SESSIONHORNS
Animator
ANIMATOR
MODE
SWING
DYNAMIC
TIMING
TEMPO


Drag MIDI song blocks here

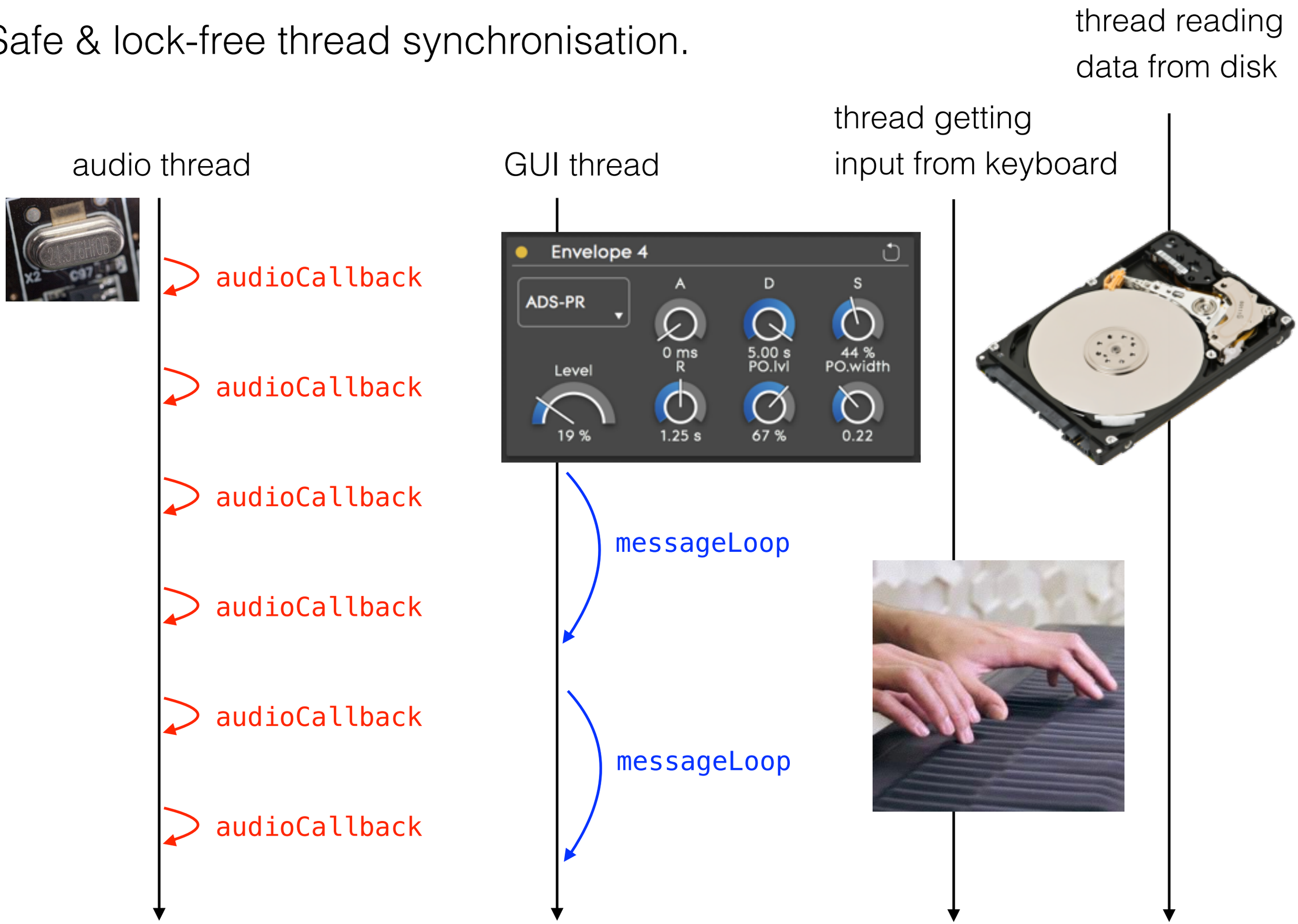
Don’t run into page faults.

- <u>solution #1</u>:
  use a low-priority thread to regularly “poke”
  the memory you will use on the audio thread
- <u>solution #2</u>:
  lock your real-time data into memory to prevent page-outs
  - Modern OSes offer APIs for that:
    `mlock()`/`munlock()` (POSIX systems),
    `VirtualLock()`/`VirtualUnlock()`(Windows)
    (nothing in the C++ standard or Boost 😟)
  - wrap calls into a smart pointer class
  - wrap calls into an STL allocator
  - pre-allocate a locked memory pool and use `boost::pool_allocator`
    or hand-written memory management

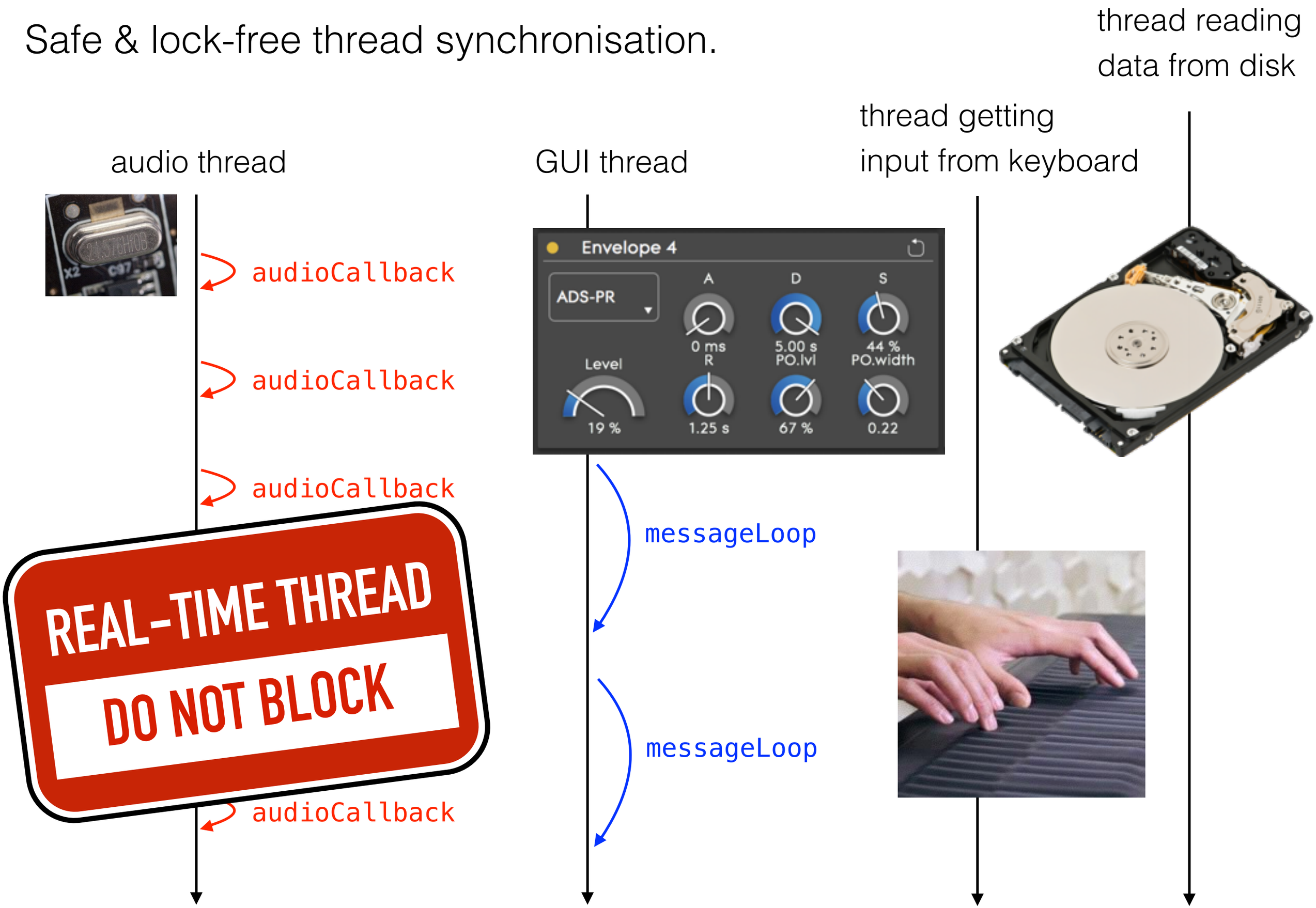
so why C++?

- most audio APIs use C or C++
- fast
- close to the metal
- allows you to use custom memory management
- supports concurrency, atomic types and lock-free programming
- allows you to write real-time safe code

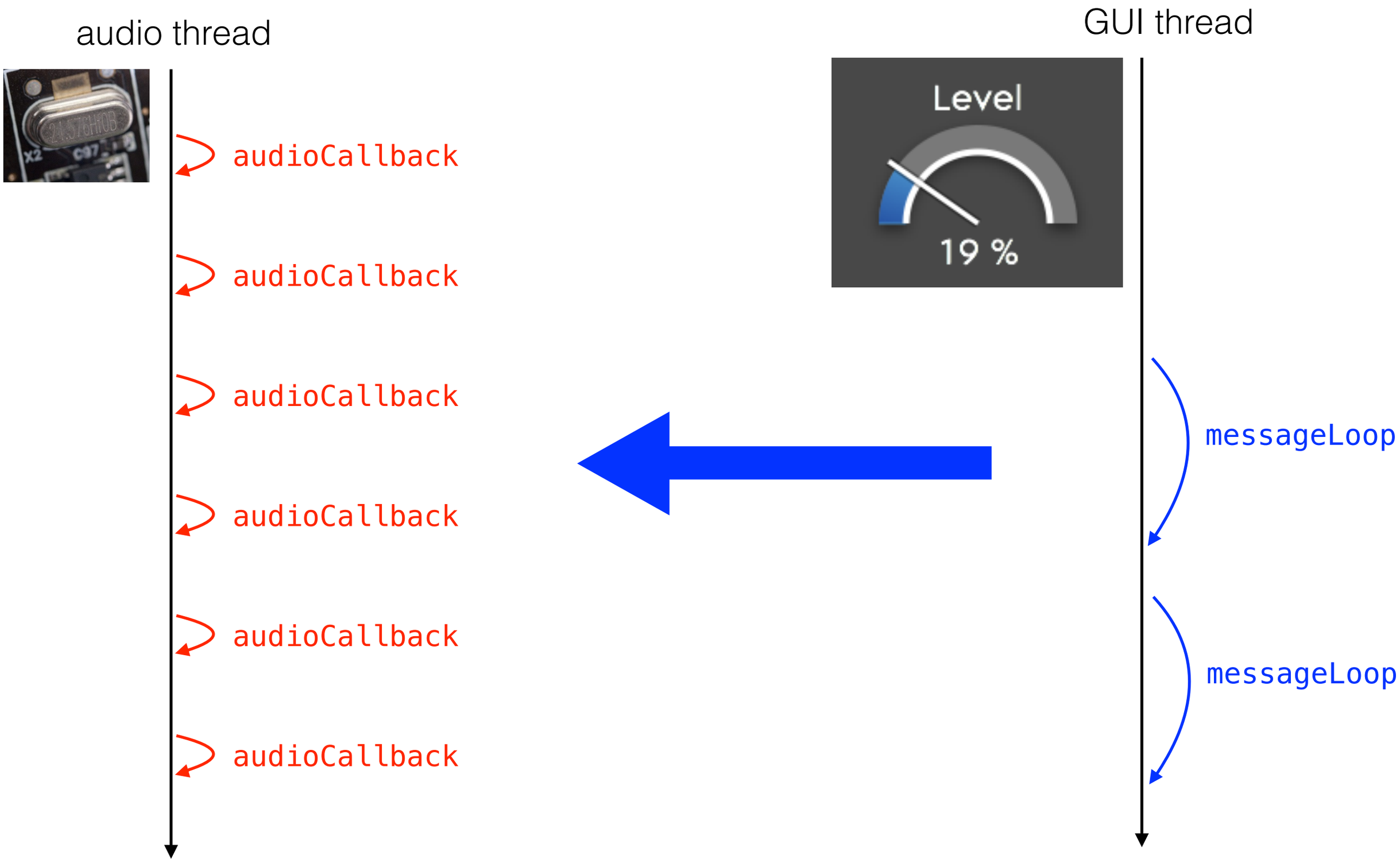
Safe & lock-free thread synchronisation.



Safe & lock-free thread synchronisation.



Safe & lock-free thread synchronisation.



```
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {
        level = newValue;
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }

    float level;
};
```

```
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {
        level = newValue;
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }

    float level;
};
```

```
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {
        level = newValue;
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }

    float level;
};
```

data race = undefined behaviour since C++11!

- Undefined behaviour
- Access is not atomic - torn reads & writes
- Compiler optimisations can break program logic
- Concurrency is not expressed in the code

Torn reads and writes

```
class NotAtomic
{
public:

    void setParameter()
    {
        parameter = 0x100000002;
    }



    std::uint64_t getParameter() const
    {
        return parameter;
    }

    std::uint64_t parameter;
};
```

Torn reads and writes

```
class NotAtomic
{
public:

    void setParameter()
    {
        parameter = 0x100000002;
    }


    std::uint64_t getParameter() const
    {
        return parameter;
    }

    std::uint64_t parameter;
};
```

compiled with gcc for 32-bit x86:

```
...
mov   DWORD PTR parameter, 2
mov   DWORD PTR parameter+4, 1
ret
...


...
mov   eax, DWORD PTR parameter
mov   edx, DWORD PTR parameter+4
ret
...
```

Breaking optimisations

```
class Foo
{
    void bar()
    {
        flag = false;

        while (flag == false)
        {
            i++;
        }
    }

    bool flag;
    int i;
};
```

Breaking optimisations

```
class Foo
{
    void bar()
    {
        flag = false;

        while (flag == false)
        {
            i++;
        }
    }

    bool flag;
    int i;
};
```

optimising compiler →

```
class Foo
{
    void bar()
    {
        flag = false;

        while (true)
        {
            i++;
        }
    }

    bool flag;
    int i;
};
```

Breaking optimisations

```
class Foo
{
    void bar()
    {
        flag = false;

        while (flag == false)
        {
            i++;
        }
    }

    volatile bool flag;
    int i;
};
```

Breaking optimisations

```
class Foo
{
    void bar()
    {
        flag = false;

        while (flag == false)
        {
            i++;
        }
    }

    volatile bool flag;
    int i;
};
```

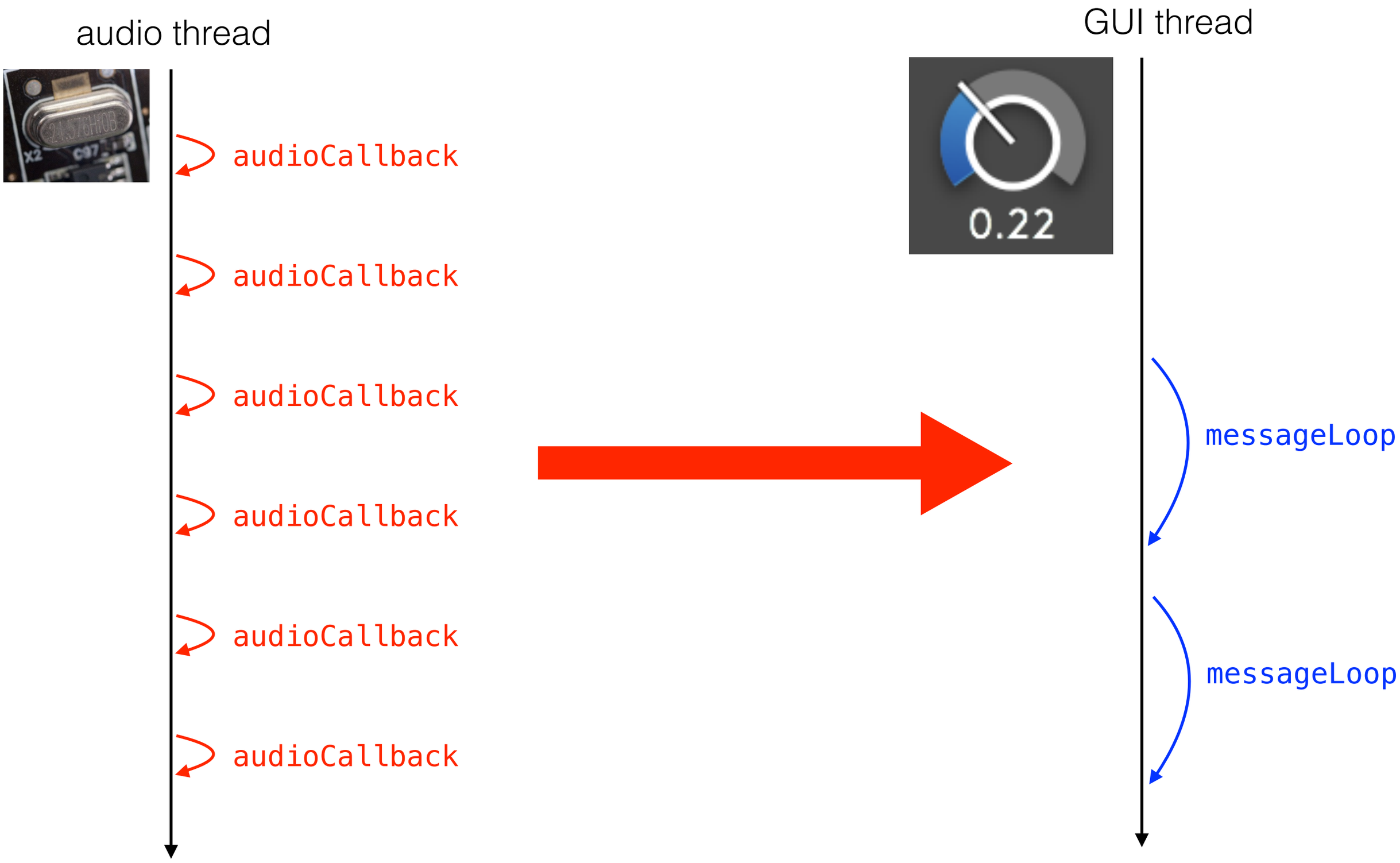
- fixes this particular problem
- but does not fix any of the other problems
- also, prevents “good” compiler optimisations
  —> wrong!!

```
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {

        level = newValue;
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {

        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }

    float level;

};
```

```
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {
        std::lock_guard<std::mutex> (m);
        level = newValue;
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {
        std::lock_guard<std::mutex> (m);
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }

    float level;
    std::mutex m;
};
```

```cpp
class Synthesiser
{
public:

    Synthesiser() : level (1.0f) {}

    // GUI thread:
    void levelChanged (float newValue)
    {

        level.store (newValue);
    }

private:

    // Audio thread:
    void audioCallback (float* buffer,
                        int numSamples) noexcept
    {

        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level.load() * getNextAudioSample();
    }

    std::atomic<float> level;

};
```

Safe & lock-free thread synchronisation.



```
class Synthesiser

    void audioCallback (float* buffer, int numSamples) noexcept
    {
        // ... some DSP ...

        parameter.store (newValue);

        updateGui();
    }

    std::atomic<float> parameter;
};
```

never do this

```
class Synthesiser

    void audioCallback (float* buffer, int numSamples) noexcept
    {
        // ... some DSP ...

        parameter.store (newValue);

        messageLoop.post (ParametersChangedNotification (parameter));
    }

    std::atomic<float> parameter;
};
```

- probably not lock-free

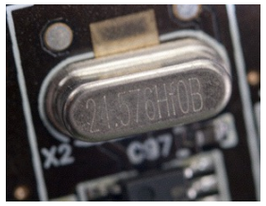
- congesting message loop with too many messages

```
class Synthesiser

    void audioCallback (float* buffer, int numSamples) noexcept
    {
        // ... some DSP ...

        parameter.store (newValue);

        guiUpToDate.store (false);
    }

    std::atomic<float> parameter;
    std::atomic<bool> guiUpToDate;

    void timerCallback()  // called 30x/second on a low priority thread
    {
        if (guiUpToDate.compare_exchange_strong (false, true))
            updateGui (parameter.load());
    }
};
```

Thread synchronisation

audio thread

GUI thread



audioCallback

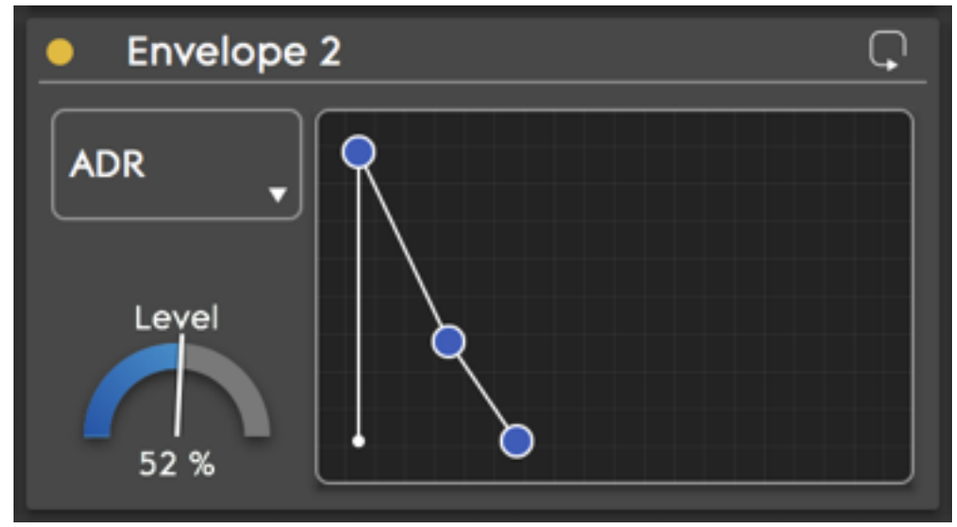
audioCallback

audioCallback

audioCallback

audioCallback

audioCallback



messageLoop

messageLoop

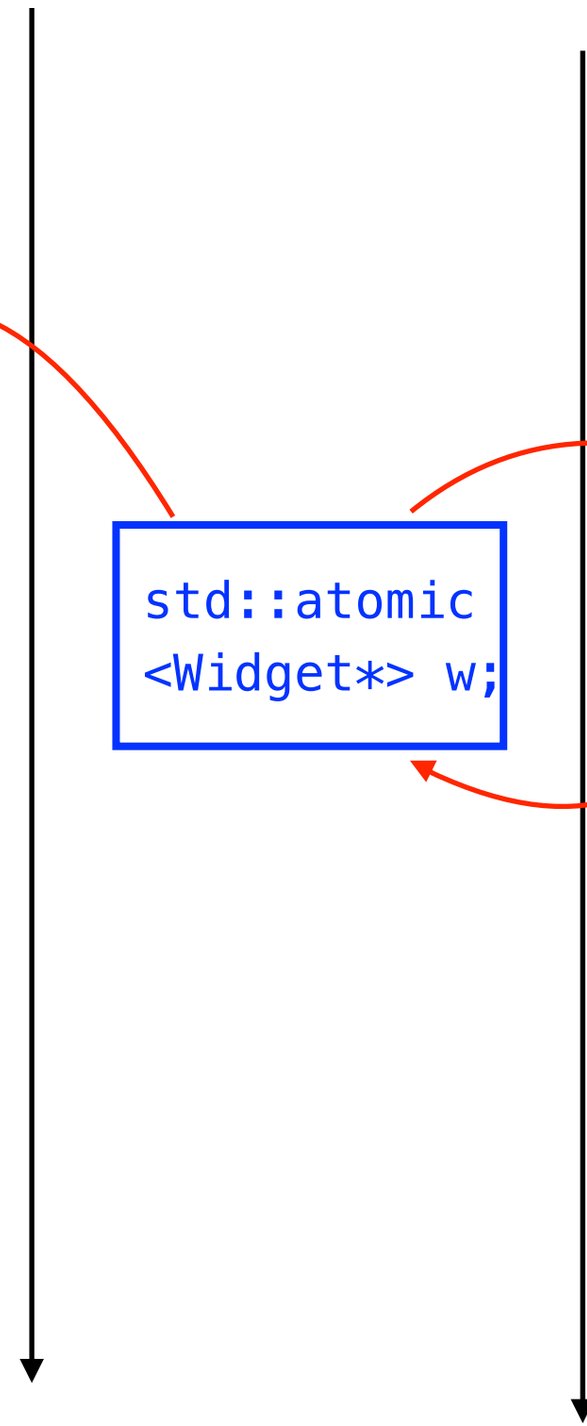
How to do this with an object?

- `std::atomic<Widget>` compiles if `std::is_trivially_copyable<Widget>::value == true`…
- …but it won't do what we want:
  - locks will be inserted by compiler
  - you can check with `std::atomic<Widget>::is_lock_free()`
- `std::atomic<Widget*>` is OK

Juggling with `std::atomic<Widget*> w` …

audio thread

```
void audioCallback (...)
{

Widget* widgetToUse;

widgetToUse = w.load();


/*
 do something
 with Widget...
 */

}
```



GUI thread

```
Widget* newValue =
    new Widget ( /* ... */ );

if (! w.compare_exchange_weak
    (oldValue, newValue))
    /* ??? */


delete oldValue;
```

~~std::atomic<std::shared_ptr<T>>~~

std::atomic_... (std::shared_ptr<T>)

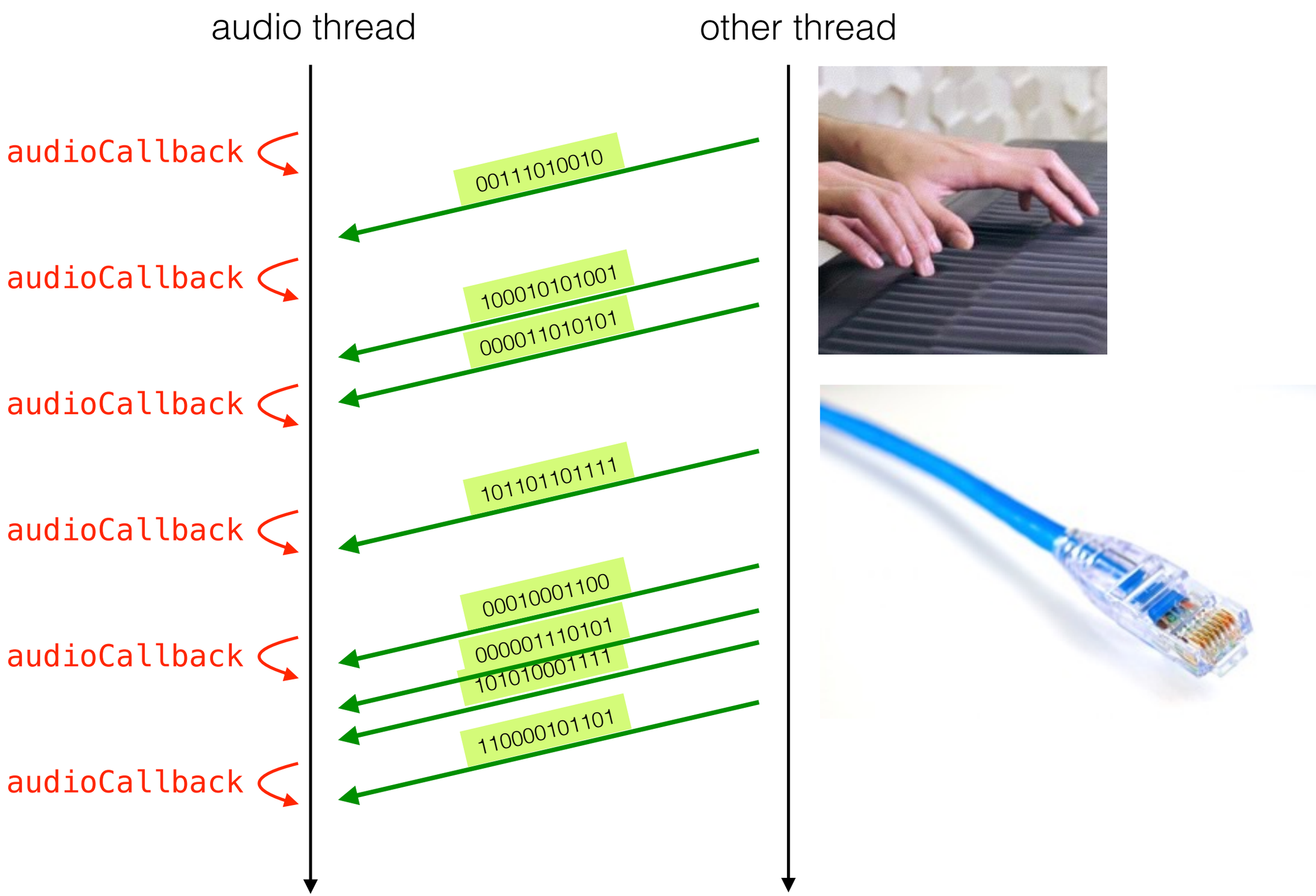
```
class Synthesiser
{
public:

    void audioCallback (float* buffer, int bufferSize)
    {
        std::shared_ptr<Widget> widgetToUse = std::atomic_load (&currentWidget);
        // do something with widgetToUse...
    }

    void updateWidget ( /* args */ )
    {
        std::shared_ptr<Widget> newWidget = std::make_shared<Widget> ( /* args */ );
        std::atomic_store (&currentWidget, newWidget);
    }

    std::shared_ptr<Widget> currentWidget;
};
```

```
class Synthesiser
{
public:

    void audioCallback (float* buffer, int bufferSize)
    {
        std::shared_ptr<Widget> widgetToUse = std::atomic_load (&currentWidget);
        // do something with widgetToUse...
    }

    void updateWidget ( /* args */ )
    {
        std::shared_ptr<Widget> newWidget = std::make_shared<Widget> ( /* args */ );
        std::atomic_store (&currentWidget, newWidget);
    }

    std::shared_ptr<Widget> currentWidget;
};
```

```
class Synthesiser
{
public:

    void audioCallback (float* buffer, int bufferSize)
    {
        std::shared_ptr<Widget> widgetToUse = std::atomic_load (&currentWidget);
        // do something with widgetToUse...
    }

    void updateWidget ( /* args */ )
    {
        std::shared_ptr<Widget> newWidget = std::make_shared<Widget> ( /* args */ );
        releasePool.add (newWidget);
        std::atomic_store (&currentWidget, newWidget);
    }

    std::shared_ptr<Widget> currentWidget;
    ReleasePool releasePool;
};
```

```
class ReleasePool : private Timer
{
public:
    ReleasePool() { startTimer (1000); }

    template<typename T> void add (const std::shared_ptr<T>& object) {
        if (object.empty())
            return;

        std::lock_guard<std::mutex> lock (m);
        pool.emplace_back (object);
    }

private:
    void timerCallback() override {
        std::lock_guard<std::mutex> lock (m);
        pool.erase(
            std::remove_if (
                pool.begin(), pool.end(),
                [] (auto& object) { return object.use_count() <= 1; } ),
            pool.end());
    }

    std::vector<std::shared_ptr<void>> pool;
    std::mutex m;
};
```

exchange data between threads

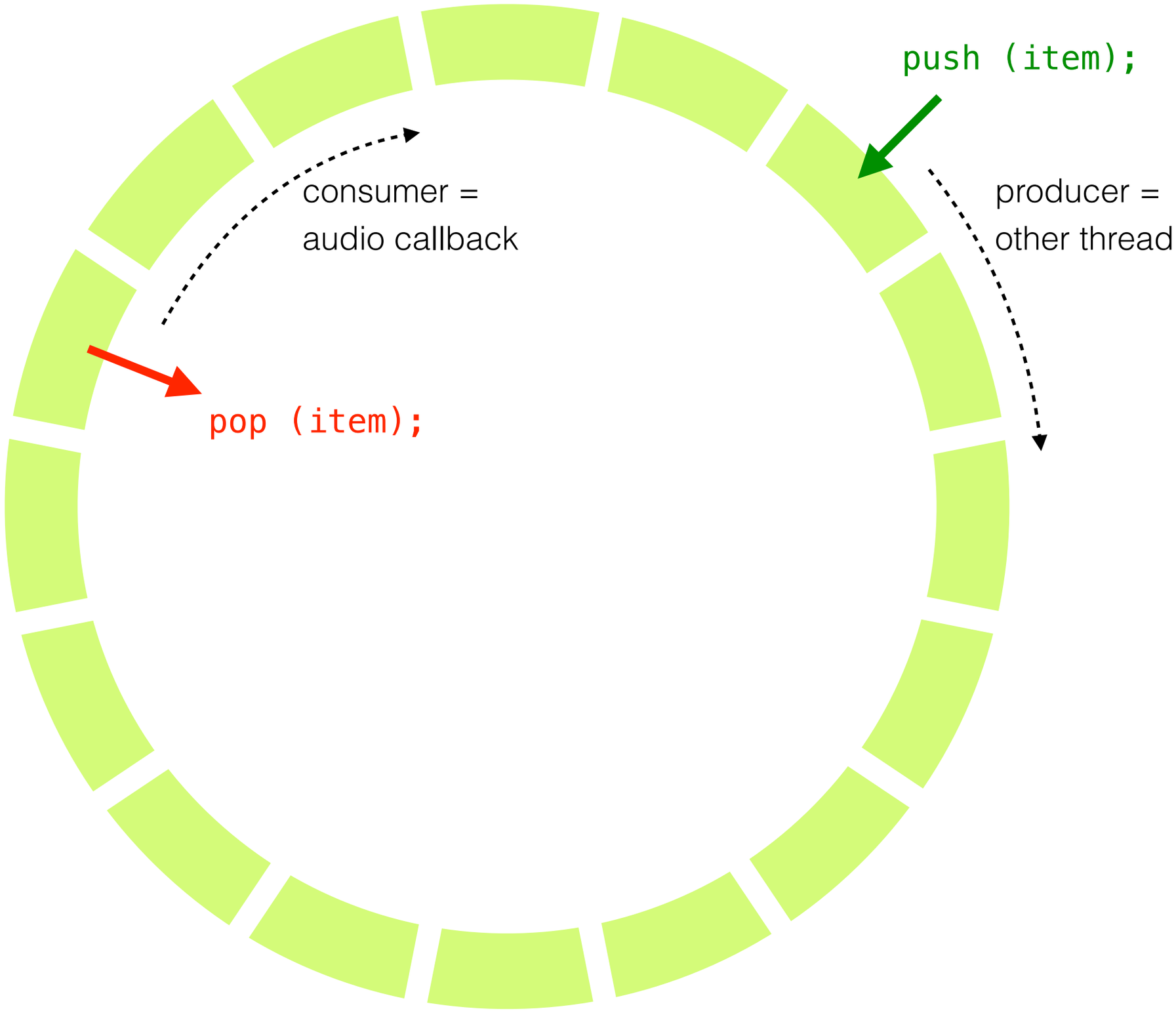


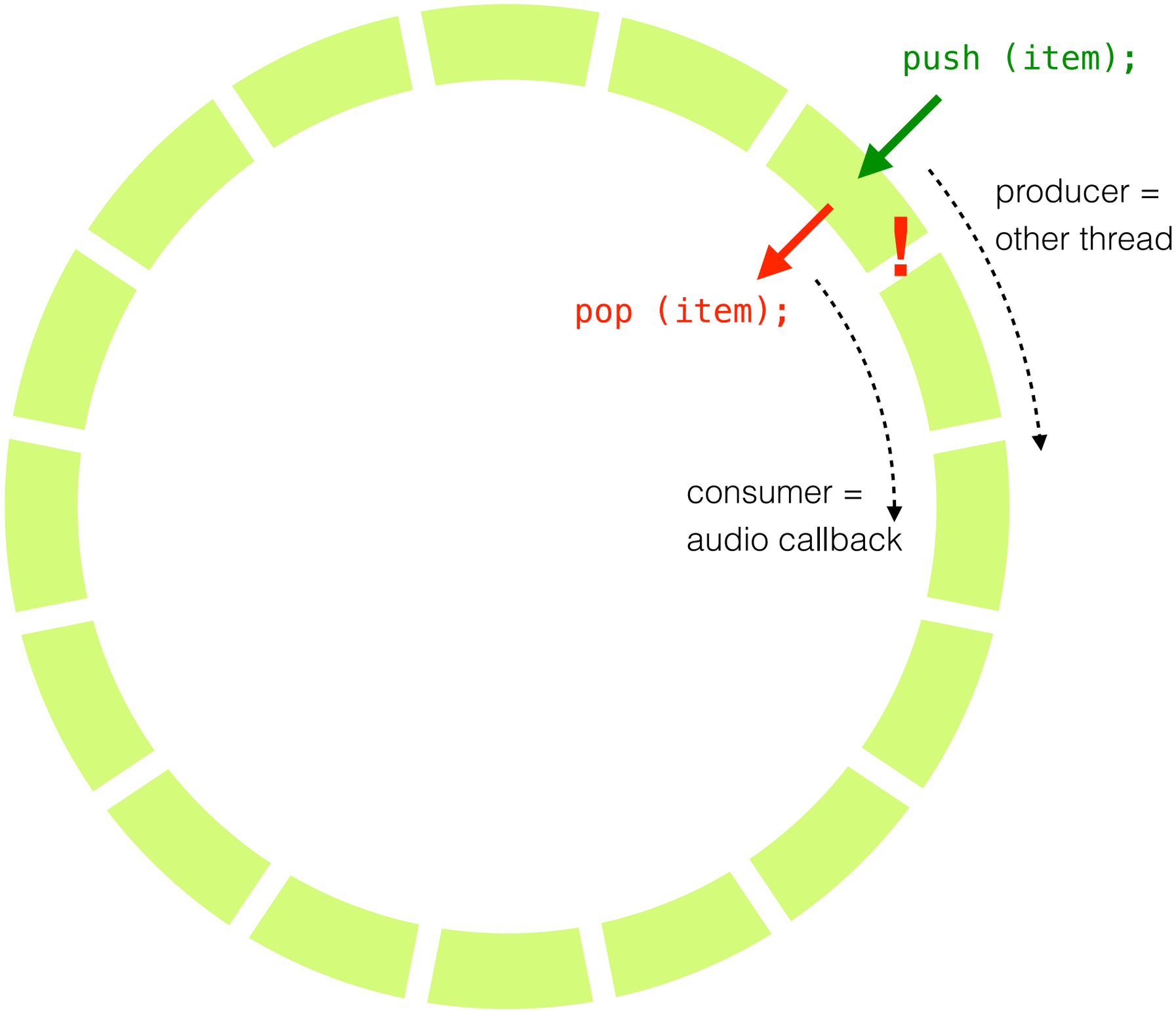
exchange data between threads

= **lock-free queue (fifo)**

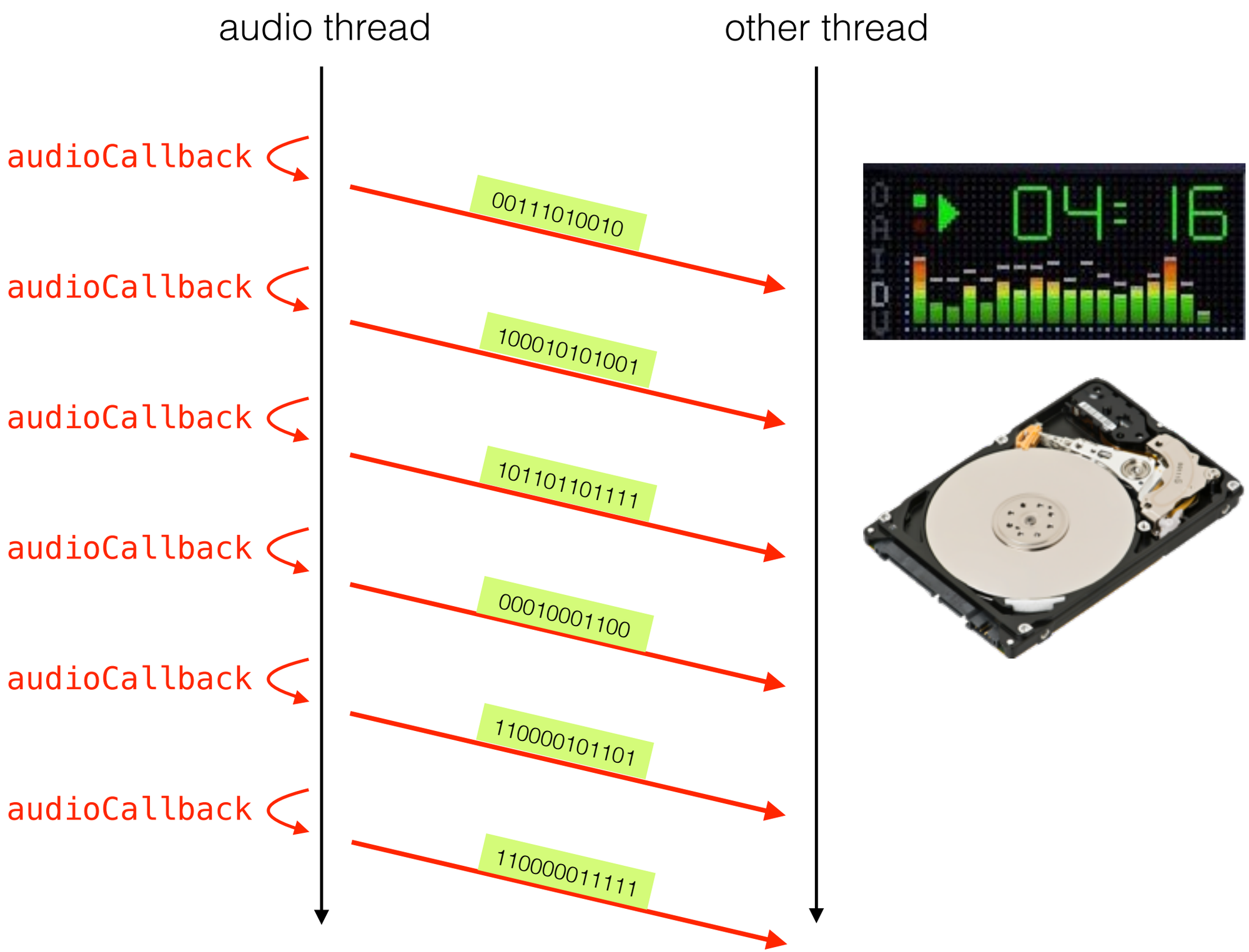
...with fixed maximum size = **lock-free ring buffer**.

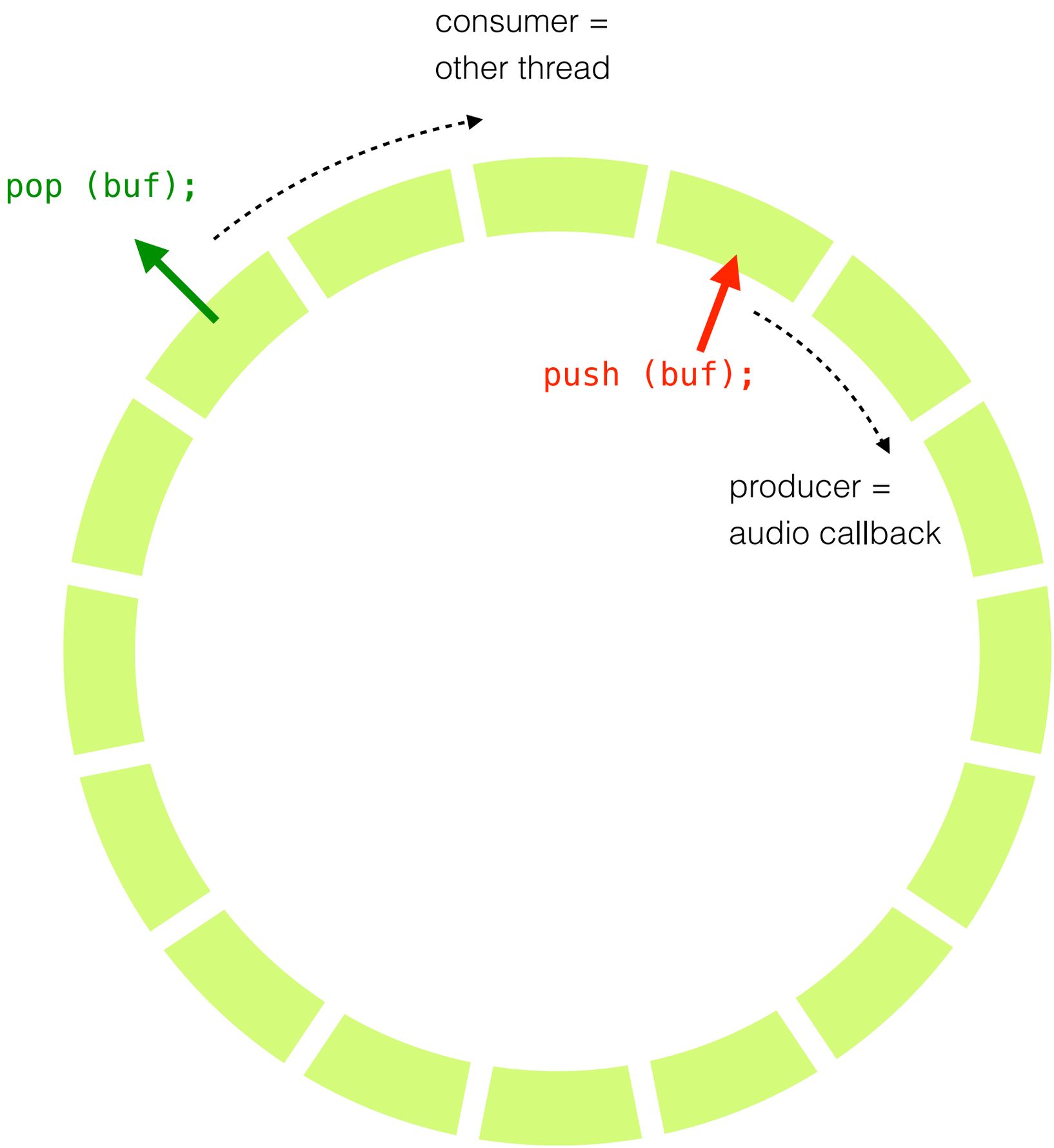
...if possible, single producer / single consumer.


push (item);
consumer =
audio callback
producer =
other thread
pop (item);

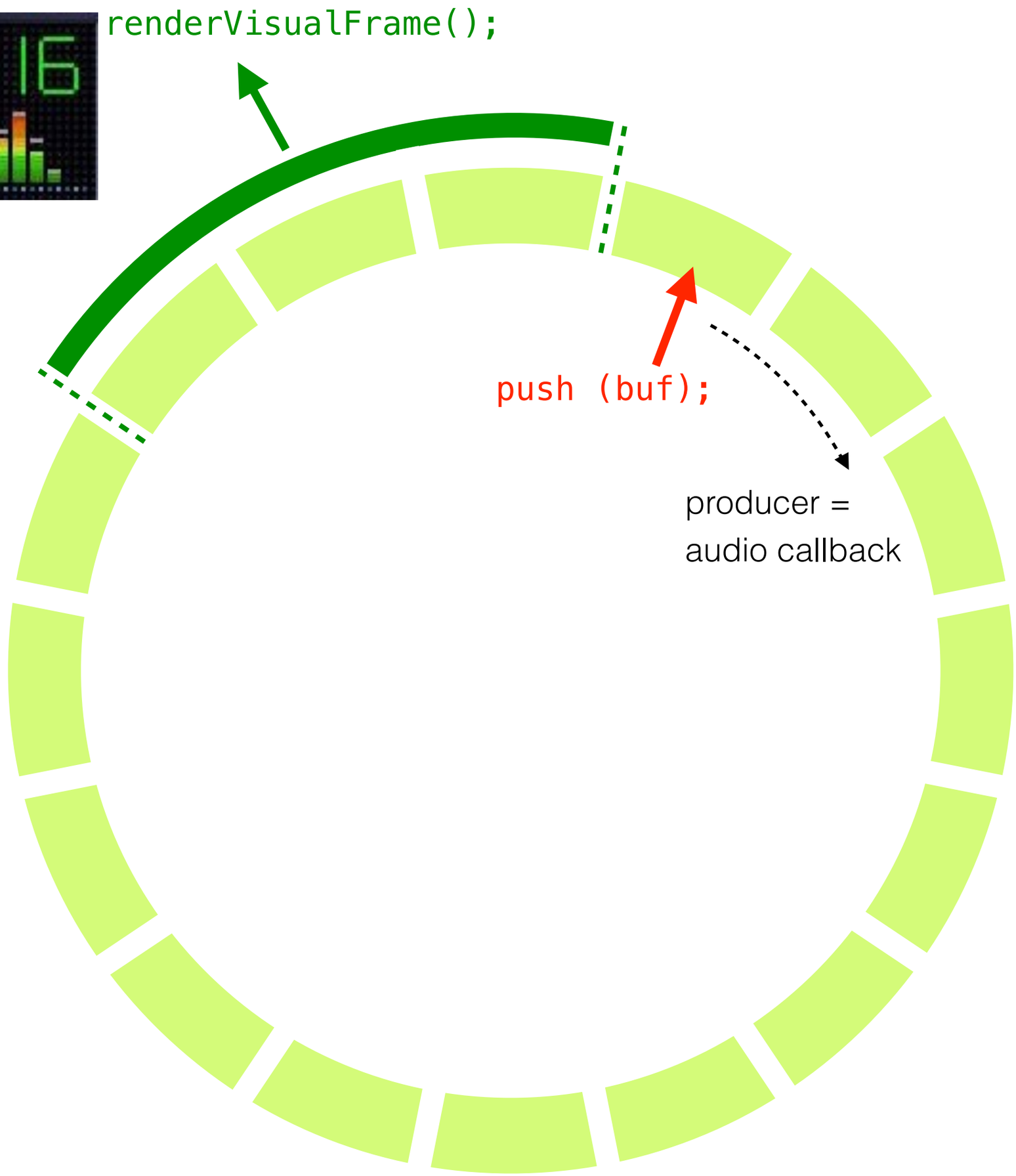

push (item);
producer =
other thread
!
pop (item);
consumer =
audio callback

exchange data between threads




consumer =
other thread
pop (buf);
push (buf);
producer =
audio callback


04:16


renderVisualFrame();
push (buf);
producer =
audio callback


to be continued...