C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

Course topics

Programming topics		Music/audio topics
Working in real time		Oscillators
Buffers and arrays		Samples
Parameter control		Wavetables
Classes and objects		Control voltages
Analog and digital I/O		Gates and triggers
Filtering		Filters
Timing in real time		Metronomes and clocks
Circular buffers		Delays and delay-based effects
State machines	Today	Envelopes
MIDI		ADSR
Block-based processing		MIDI
Threads		Additive synthesis
Fixed point arithmetic		Phase vocoders
ARM assembly language		Impulse reverb

Lecture 11: Circular buffers

What you'll learn today:

Keeping track of previous audio samples
Why and how to use circular buffers
Creating an audio delay

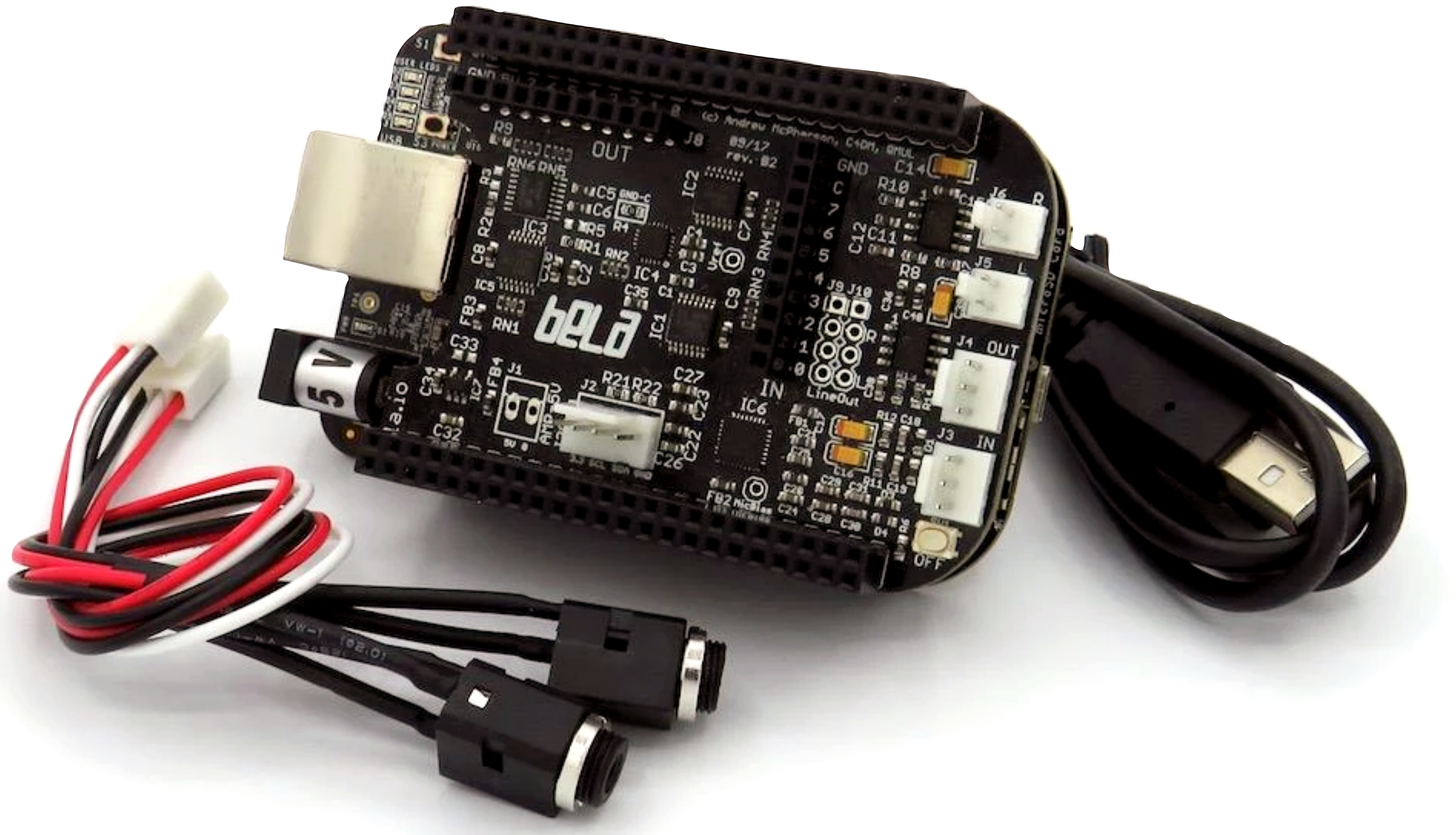
What you'll make today:

An adjustable delay and an echo effect

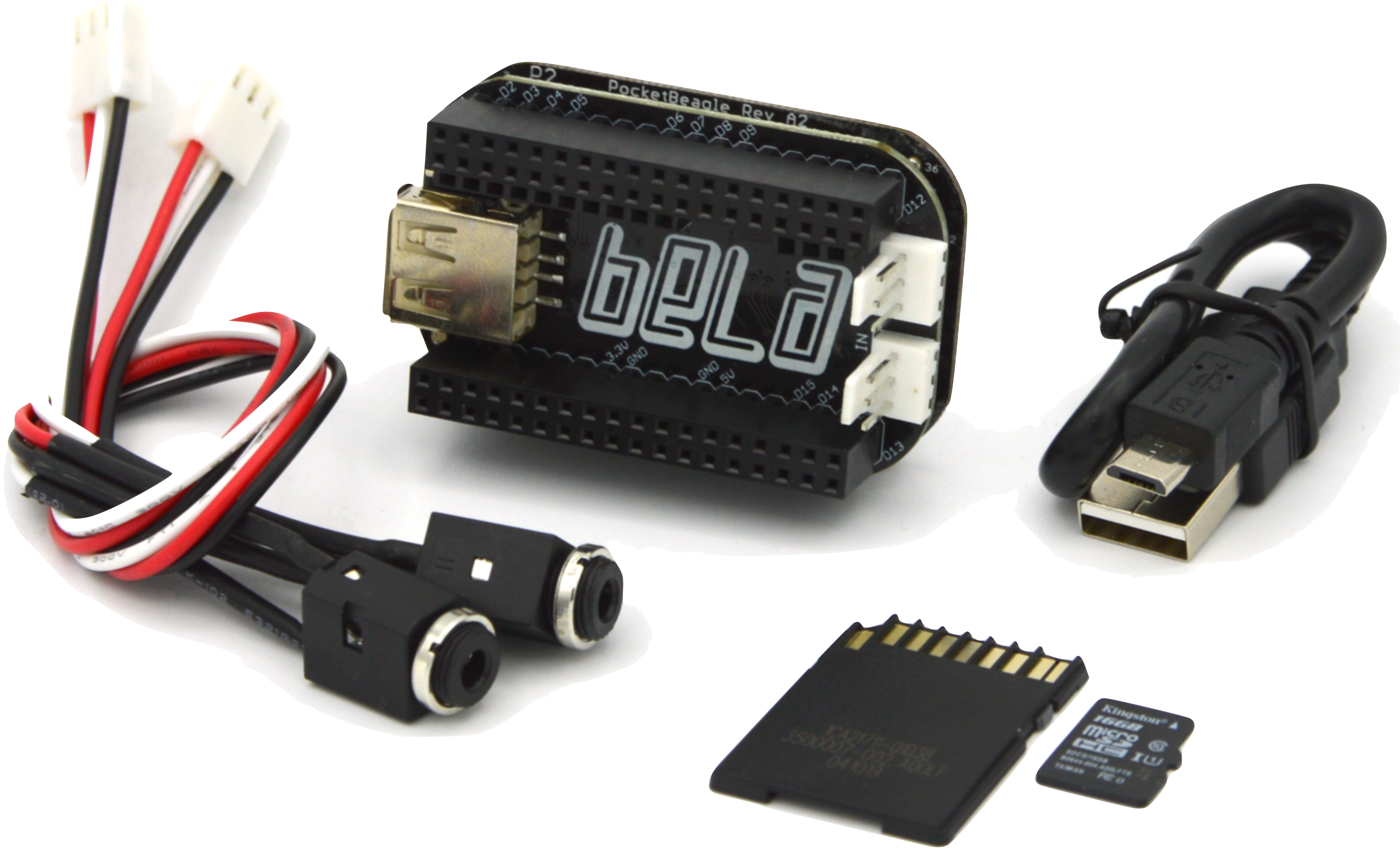
Companion materials:

github.com/BelaPlatform/bela-online-course

What you'll need



or

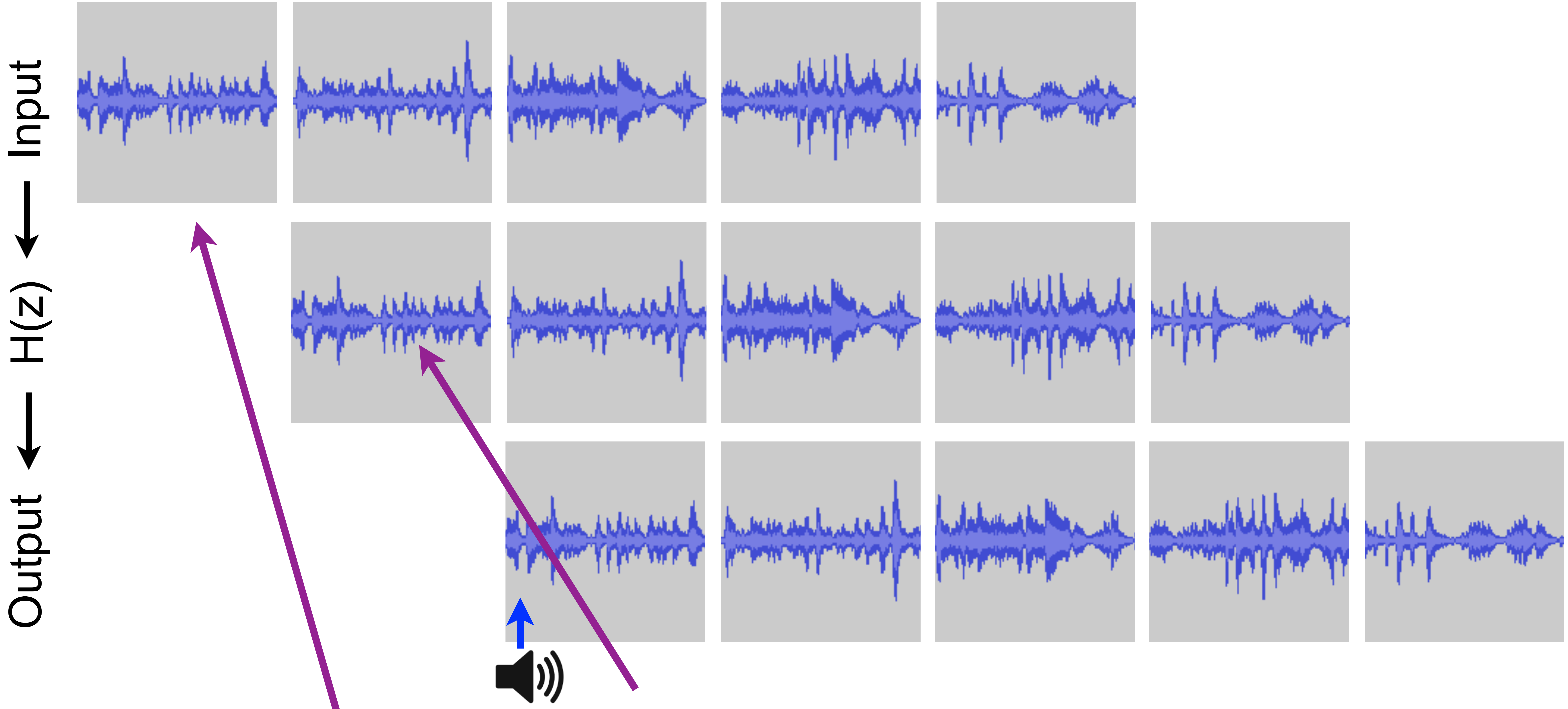


Bela Starter Kit

[shop.bela.io]

Bela Mini Starter Kit

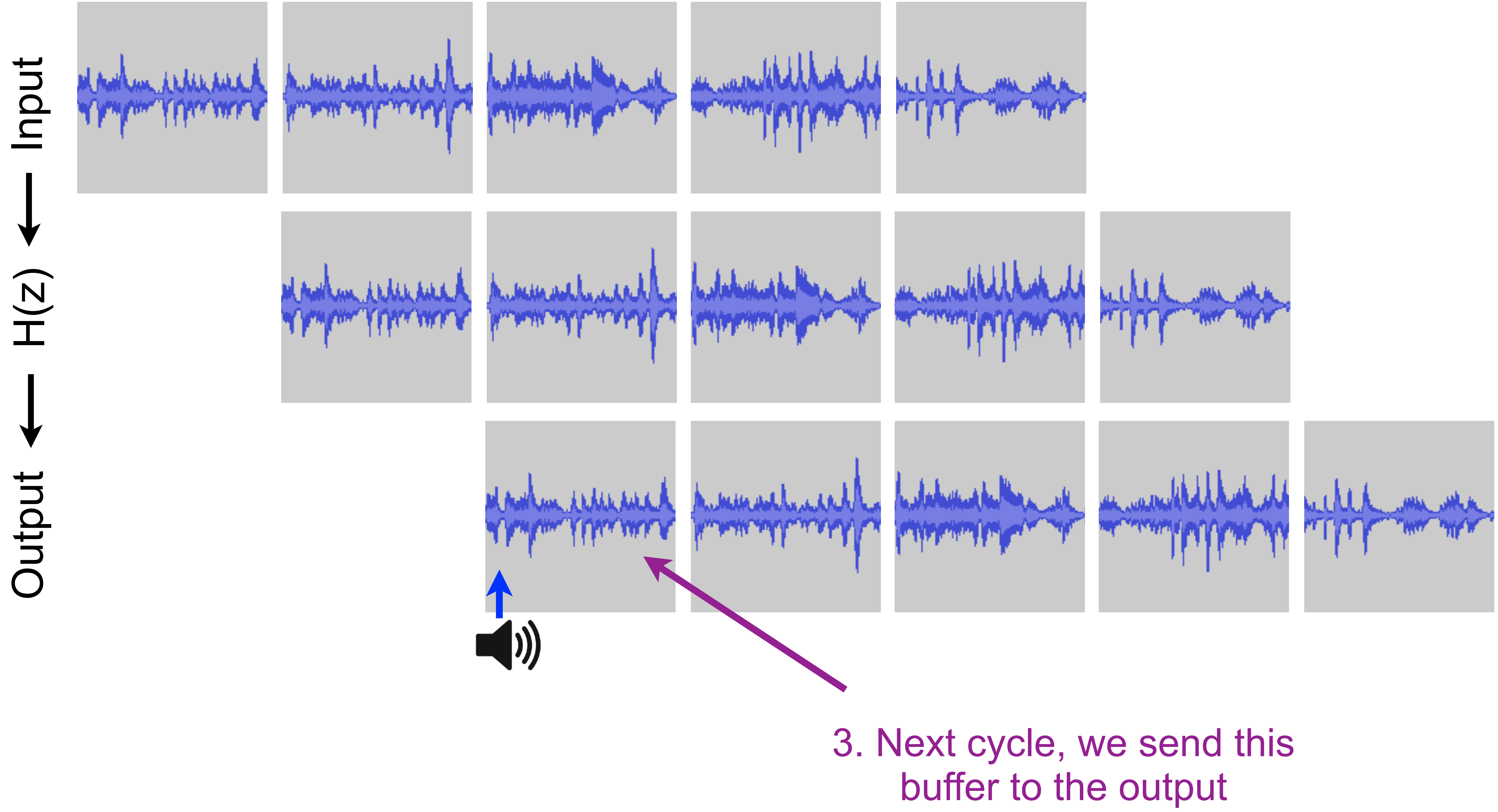
Review: audio buffering



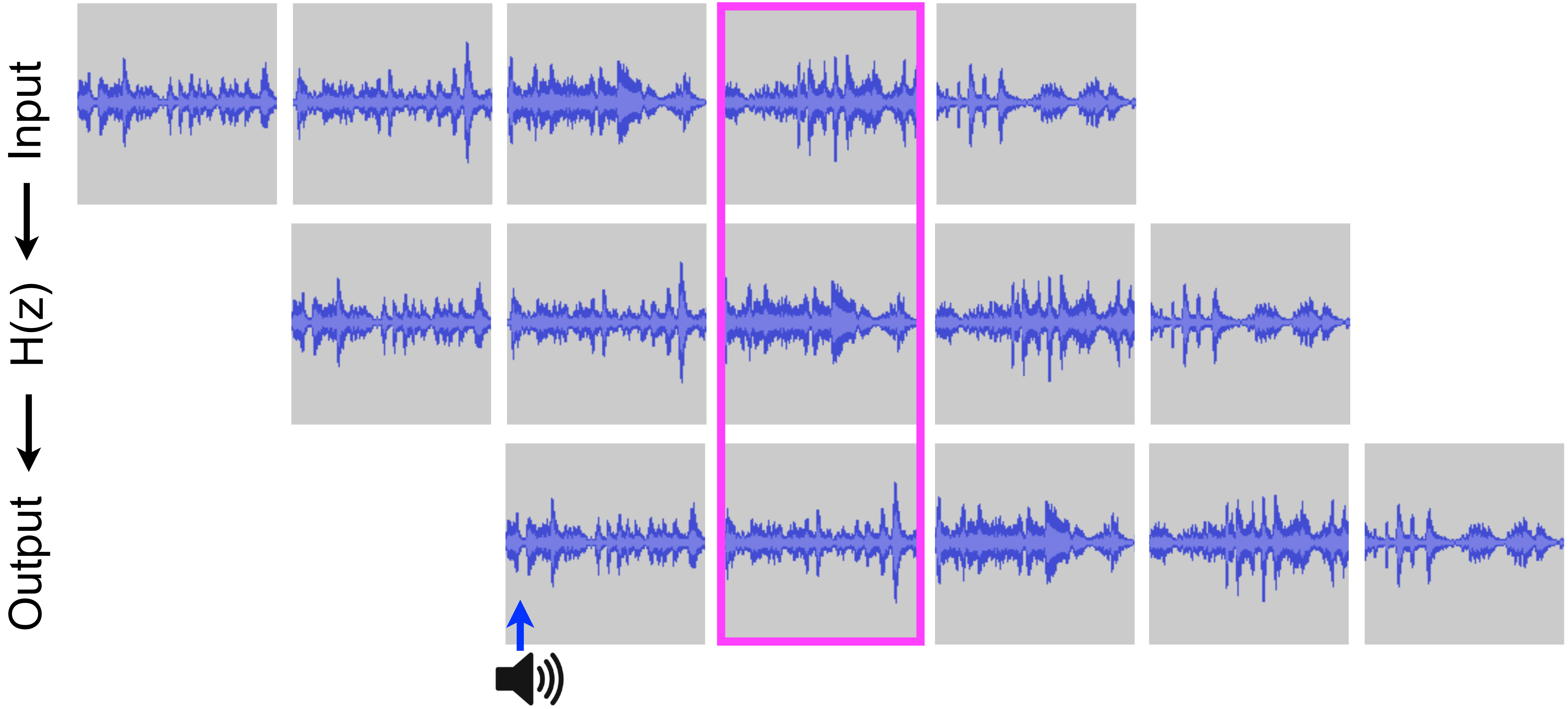
1. First we fill up
a buffer of samples

2. We process this buffer
while the next one fills up

Review: audio buffering

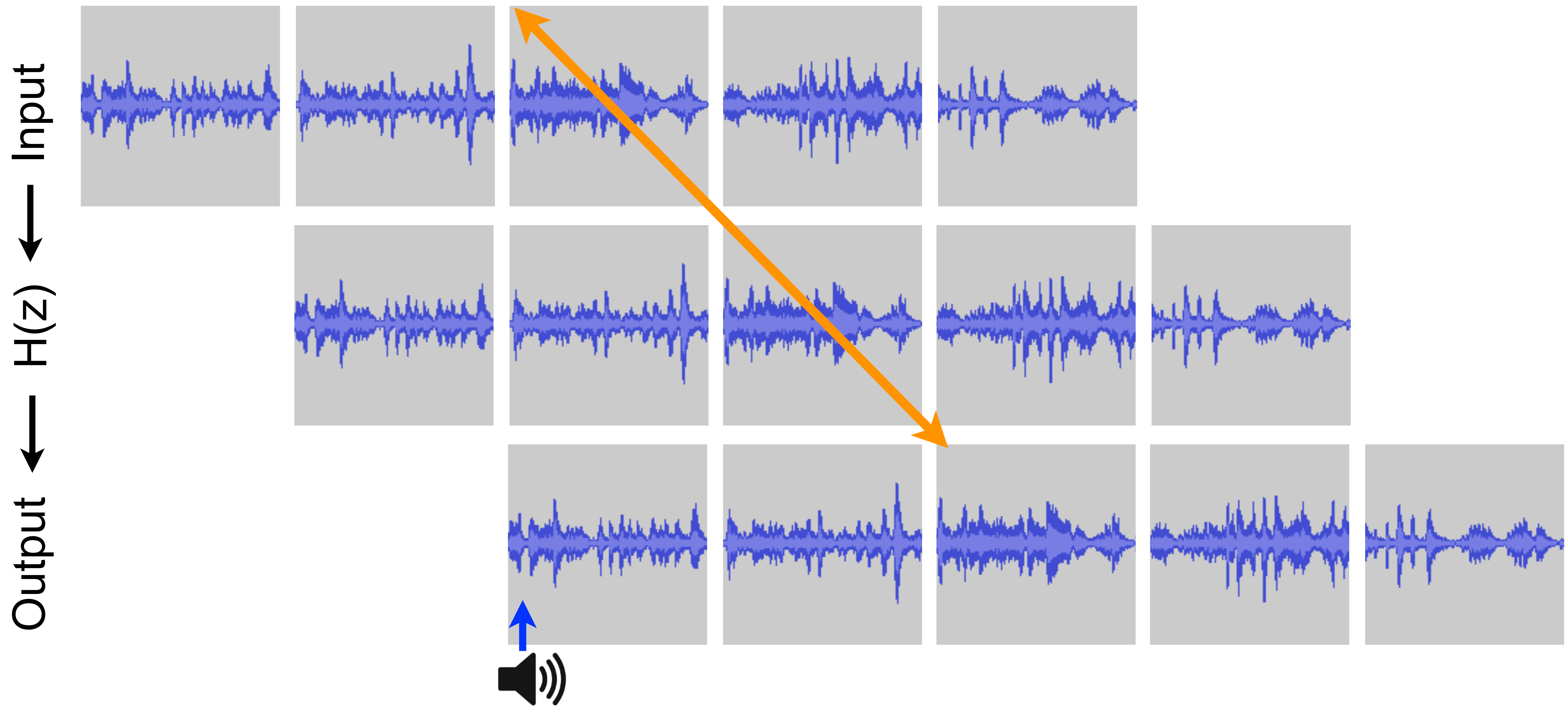


Review: audio buffering



At any given time, we are reading from ADC, processing a block, and writing to DAC

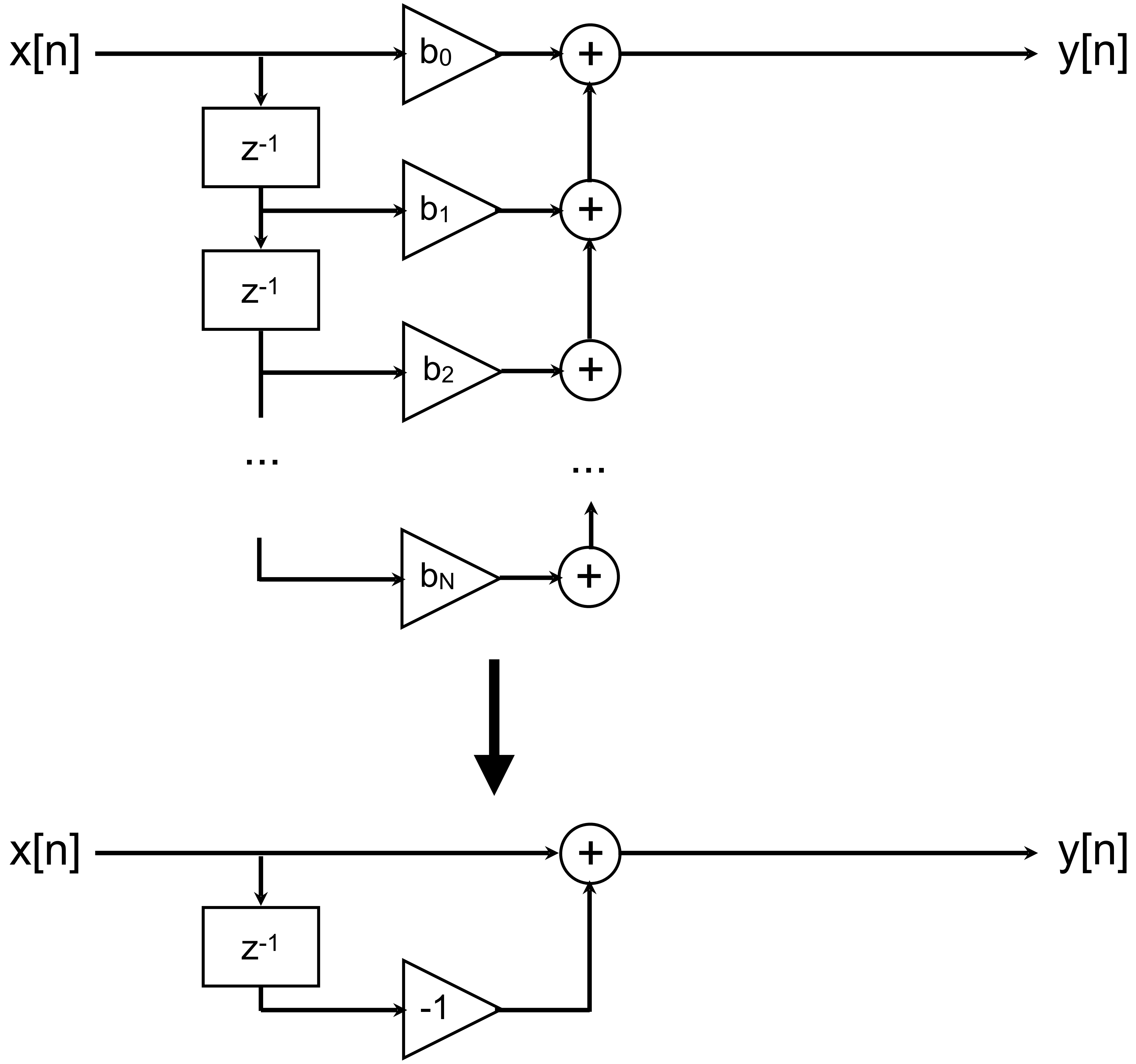
Review: audio buffering



Total latency is 2x buffer length

Review: a simple filter

- We want to implement this filter: $y[n] = x[n] - x[n-1]$
- Block diagram for an FIR filter:
- What are the coefficients b_n?
  - $b_0 = 1$
  - $b_1 = -1$
- What information do we need to keep track of?
  - Previous value of $x[n]$
  - Use a global variable



Saving a previous input

Calculating the filter $y[n] = x[n] - x[n-1]$

```
float gLastSample = 0;
```

global variable to hold *x*[*n* - 1]

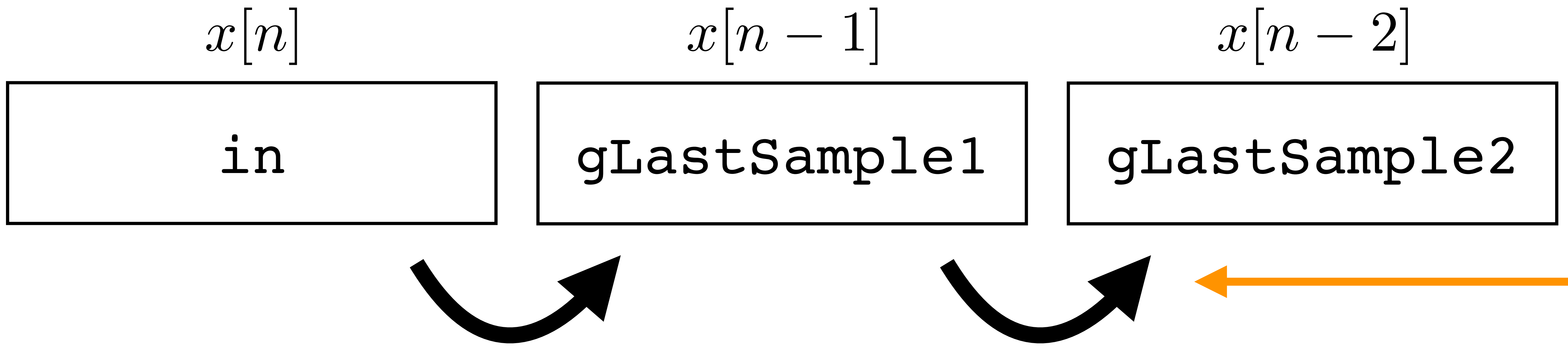
```
for(unsigned int n = 0; n < context->audioFrames; n++) {
    // ...let's say "in" holds our input, calculated somehow:
    float in =

    // Here we implement a first-order FIR filter:
    // y[n] = x[n] - x[n-1]
    float out = in - gLastSample;
    gLastSample = in;
}
```

so that the **next** time it holds x[n-1]

set `gLastSample` to x[n]...

Saving 2 previous inputs

- Say we want to calculate $y[n] = x[n] + x[n-1] + x[n-2]$
- How do we save 2 previous inputs?
  - Need two global variables (we'll call them `gLastSample1` and `gLastSample2`)
  - Update them at the end of the loop, like before:



Important! This assignment has to happen first. (Why?)

```
// Here we implement a second-order FIR filter:
// y[n] = x[n] + x[n-1] + x[n-2]
float out = in + gLastSample1 + gLastSample2;
gLastSample2 = gLastSample1;
gLastSample1 = in;
```

- Do we still need `gLastSample1` here? $y[n] = x[n] + x[n-2]$

Saving many previous inputs

- Let's extend the previous concept to saving the last 100 samples:

```
float gLastSample1;
float gLastSample2;
float gLastSample3;
float gLastSample4;
float gLastSample5;
float gLastSample6;
float gLastSample7;
float gLastSample8;
float gLastSample9;
float gLastSample10;
float gLastSample11;
float gLastSample12;
float gLastSample13;
float gLastSample14;
float gLastSample15;
float gLastSample16;
float gLastSample17;
float gLastSample18;
float gLastSample19;
float gLastSample20;
float gLastSample21;
float gLastSample22;
float gLastSample23;
float gLastSample24;
float gLastSample25;
float gLastSample26;
float gLastSample27;
float gLastSample28;
float gLastSample29;
float gLastSample30;
float gLastSample31;
float gLastSample32;
```

Saving many previous inputs

- Let's extend the previous concept to saving the last 100 samples:
- Better plan: use an array for the previous samples

```
float gLastSamples[100] = {0};
```

Simple way to initialise all array elements to 0

  - Let's define the indices like this: `gLastSamples[k]` ⟷ $x[n-1-k]$
    - So for example, `gLastSamples[0]` corresponds to $x[n-1]$
  - Where would we find $x[n-100]$? `gLastSamples[99]`
  - What does `gLastSamples[37]` hold? $x[n-38]$
  - What does `gLastSamples[100]` hold?
    - Nothing! Array has only 100 elements, so valid indices are 0 to 99
- Task: write some pseudocode to save the last 100 samples
  - Implement the equation $y[n] = x[n-100]$

Saving many previous inputs

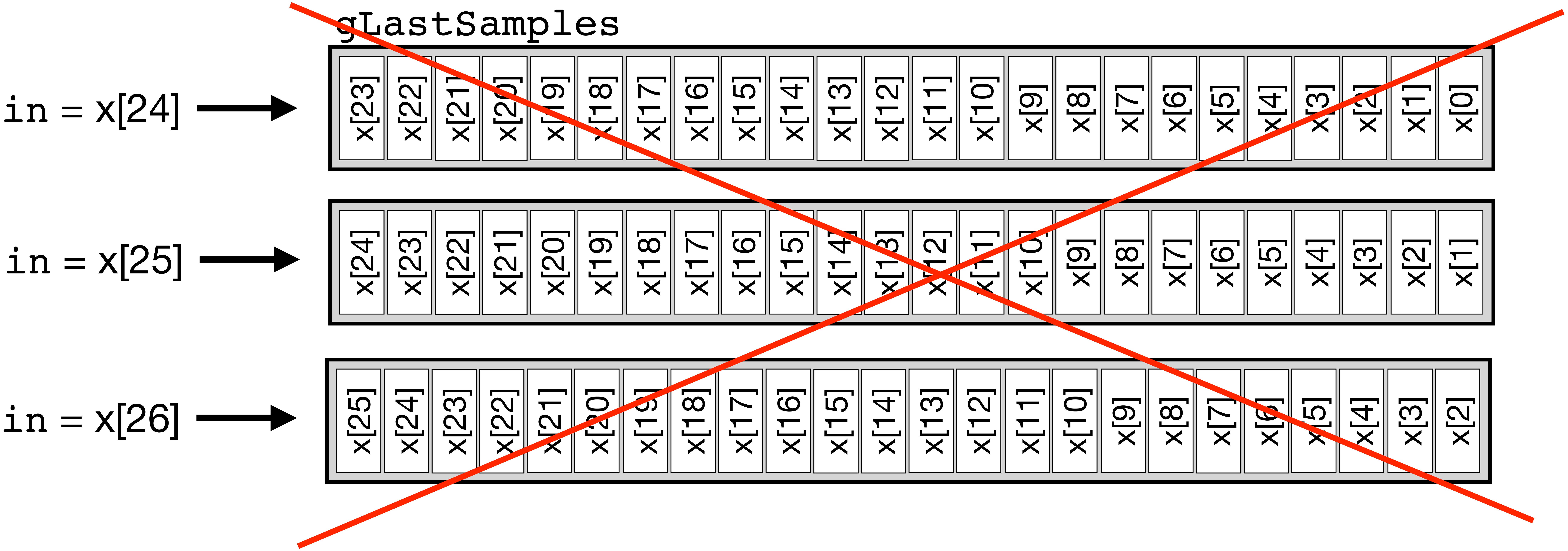
$$y[n] = x[n - 100] \longrightarrow$$

```
float gLastSamples[100] = {0};

for(int n = 0; n < context->audioFrames; n++) {
    float in = 
    float out = gLastSamples[99];

    // Move every sample back one element in the array
    // Notice: have to start from back
    for(int i = 99; i > 0; i--)
        gLastSamples[i] = gLastSamples[i - 1];

    // First element in the array is the most recent input sample
    gLastSamples[0] = in;
}
```

- Notice use of internal `for()` loop
  - Also notice its direction: decreasing
  - Why `i > 0` and not `i >= 0` ?
    - Can't access `gLastSamples[-1]`
- What is a drawback of this whole approach? Inefficiency!

Moving samples

- Moving memory around is wasteful!
  - In this illustration, we are saving the last 24 samples:



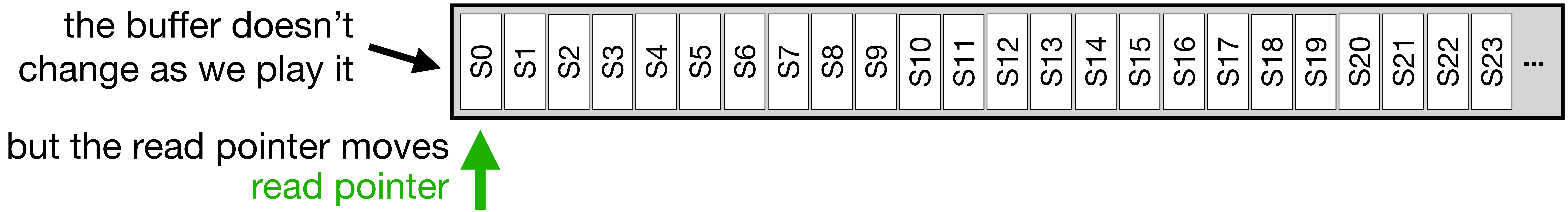
- Bad idea: don’t move 23 samples to add 1

Circular buffering

- Instead, leave the old samples in place when we add new ones
- This is called a circular buffer: a memory buffer that acts like a loop
  - Write each new sample in the next location, from beginning to end
  - When we get to the end, go back to the beginning again
  - Keep track of the write pointer, which tells us which slot we write to next

---

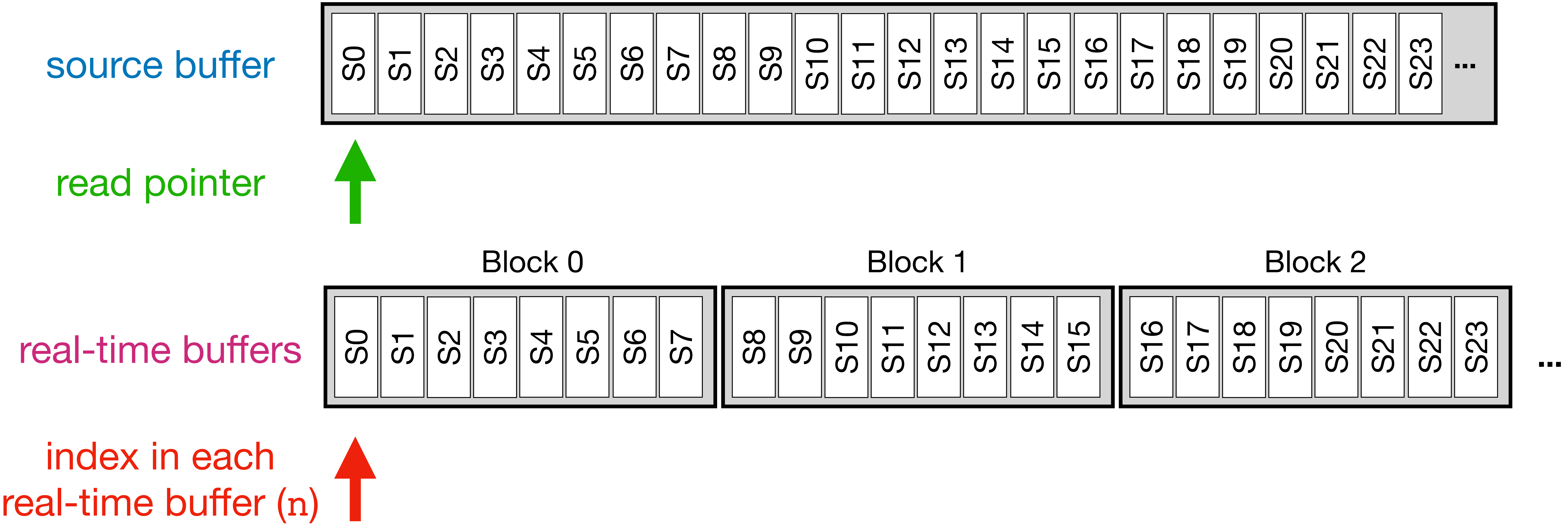
- Before getting into those details, let's review reading from a buffer...
  - The read pointer was a global variable that kept track of which sample we were reading



Review: indexing

- One of the hardest parts of working with buffers can be keeping track of what each index means
- In this case, we've got two different kinds of buffers to think about:
  1. The recorded sound (let's call it the source buffer)
     - Only one buffer whose contents don't change
     - Length: number of samples in the source sound (possibly long)
  2. The buffer for each real-time audio block
     - A new buffer each time `render()` is called, accessed via `audioWrite()`
     - Length: block size of the real-time system (e.g. 16)
- Therefore, we need to keep track of two indexes:
  1. Where are we playing in the source buffer? (read pointer or play head)
  2. Where are we writing in the output buffer? (starts over from 0 each block)

Review: indexing



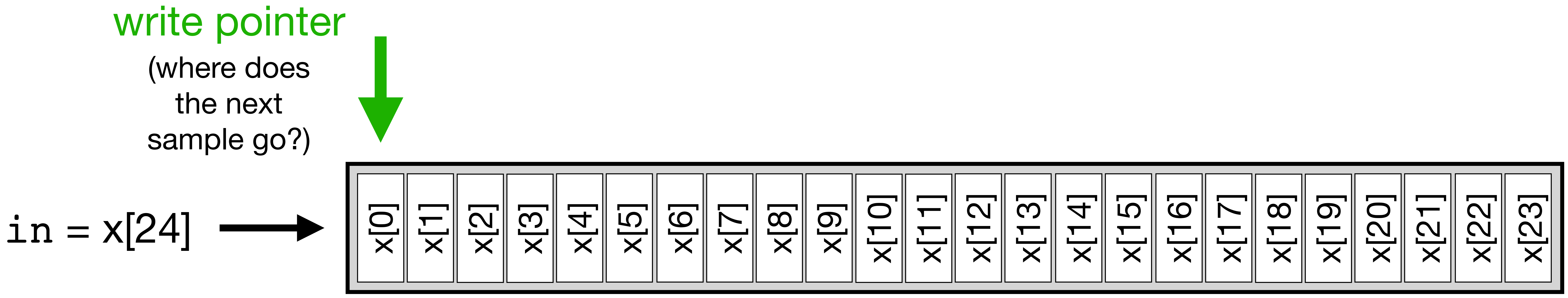
`render()` [1st time]
`n` goes from 0 to 7
`gReadPointer` goes from 0 to 7

`render()` [2nd time]
`n` goes from 0 to 7
`gReadPointer` goes from 8 to 15

`render()` [3rd time]
`n` goes from 0 to 7
`gReadPointer` goes from 16 to 23

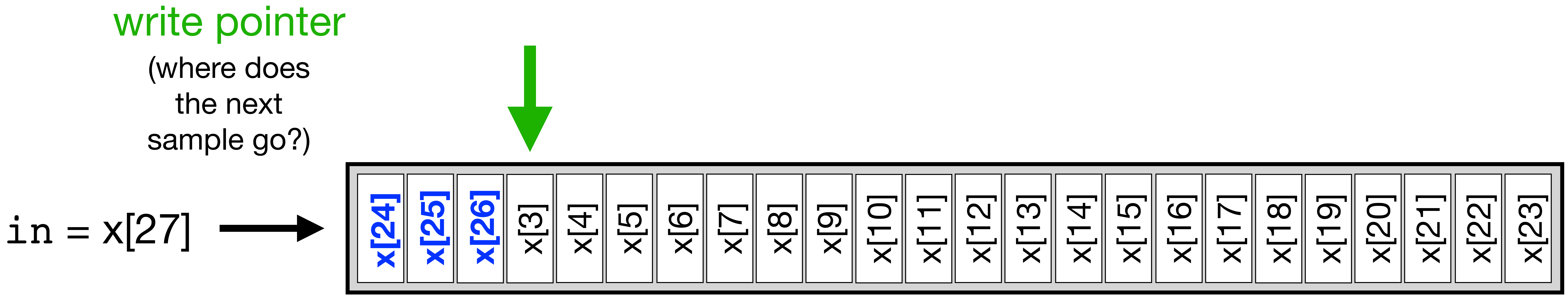
Circular buffering

- Back to writing to our circular buffer:
- A circular buffer is a memory buffer (array) that acts like a loop
  - Write each new sample in the next location, from beginning to end
  - When we get to the end, go back to the beginning again
  - Keep track of the write pointer, which tells us which slot we write to next
- The buffer always ends up holding the N most recent samples
  - We just need to keep track of which sample is held where



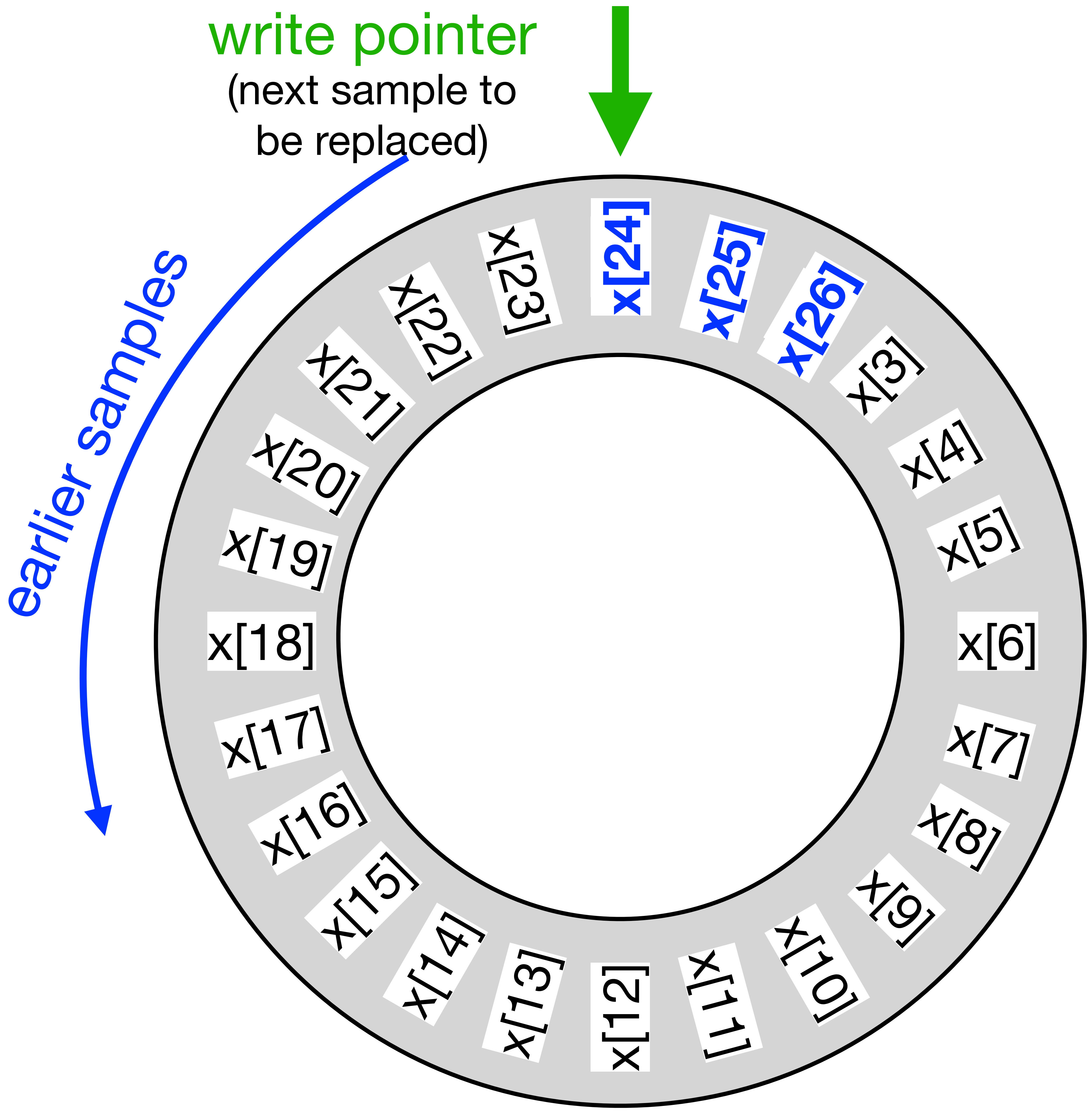
Circular buffering

- Back to writing to our circular buffer:
- A circular buffer is a memory buffer (array) that acts like a loop
  - Write each new sample in the next location, from beginning to end
  - When we get to the end, go back to the beginning again
  - Keep track of the write pointer, which tells us which slot we write to next
- The buffer always ends up holding the N most recent samples
  - We just need to keep track of which sample is held where



Circular buffering

- Another equivalent view:
- The write pointer tells us where to find the front of the buffer
  - Points just past the most recent sample
    - i.e. it's the oldest sample in the buffer until it's replaced
  - To find earlier samples, look backward from the write pointer
- At any given time:
  - The buffer holds the N most recent samples
  - Each individual sample never moves until it is eventually replaced



Write pointer

- The circular buffer has two components:
  1. A region in memory (array) to store the samples
  2. A write pointer to keep track of where we are
- Remember, there is no functional beginning or end to a circular buffer!
- In code, we need to declare two (global) variables:

```
std::vector<float> gDelayBuffer;
unsigned int gWritePointer = 0;
```

  - When we have a new sample, store it at the write pointer, then increment the pointer

```
gDelayBuffer[gWritePointer] = in;
gWritePointer++;
```

  - What else do we need to do?
    - Keep the write pointer in range

```
if(gWritePointer >= gDelayBuffer.size())
    gWritePointer = 0;
```

Circular buffer task

- Using the circular-buffer code example from the companion materials
- Task: implement a 0.5-second delay on only the left channel
  - The right channel should have no delay, so the difference can be clearly heard
- You will need to:
  - Declare variables for the buffer (use `std::vector<float>`) and the write pointer
  - Allocate the buffer to hold 0.5 seconds (see the `std::vector::resize()` method)
  - Initialise the write pointer in a sensible place (e.g. at 0)
  - Read samples out of the buffer which are 0.5 seconds old
  - Store samples in the buffer as they come in, and move the write pointer
- Hint: the write pointer always points to the oldest sample in the buffer
  - If you set your buffer size correctly, you only need a single pointer!

Circular buffer code

```
std::vector<float> gDelayBuffer;
unsigned int gWritePointer = 0;

bool setup(BelaContext *context, void *userData)
{
    // [...]
    // Allocate the circular buffer to 0.5 seconds
    gDelayBuffer.resize(0.5 * context->audioSampleRate);
    return true;
}

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        float in = 

        // Read the output from the write pointer (oldest sample)
        float out = gDelayBuffer[gWritePointer];

        // Overwrite the buffer at the write pointer, the increment and wrap pointer
        gDelayBuffer[gWritePointer] = in;
        gWritePointer++;
        if(gWritePointer >= gDelayBuffer.size())
            gWritePointer = 0;

        // Write the input and output to different channels
        audioWrite(context, n, 0, in);
        audioWrite(context, n, 1, out);
    }
}
```

Indexing in a circular buffer

- The circular buffer isn't literally a circle in the computer's memory
  - If we fall off the end, we need to wrap the index around to the beginning
- Suppose our buffer is `gDelayBuffer` and is 100 samples long
  - What index is 2 samples older than `gDelayBuffer[51]`?
    - `gDelayBuffer[49]`
  - What index is 2 samples older than `gDelayBuffer[1]`?
    - `gDelayBuffer[99]` **(why?)**
  - What index is 5 samples older than `gDelayBuffer[0]`?
    - `gDelayBuffer[95]`
  - How do I find 100 samples older than `gDelayBuffer[10]`?
    - **Can't!** If a given sample is in the buffer, then there are only 99 more stored there.
- What is the generic way of doing this?
  - Modulo arithmetic

Modulo arithmetic

- `x % y` ("x mod y") gives the remainder after `x` is divided by `y`
  - If `x > 0` and `y > 0`, then the range of `x % y` is 0 to y-1
    - For example: 5 % 2 = 1
- Modulo arithmetic completes the "circle" in the circular buffer
  - It lets us always stay in the right range of array indices
  - It wraps around when we give it an index off the end of the buffer
- How do we use modulo arithmetic to implement a circular buffer?
  - What is the value of `y` in the expression above?
    - The buffer size
  - `gDelayBuffer[(n + 2) % 100];` ⟶ 2 samples forward (later) in buffer
  - `gDelayBuffer[(n - 2) % 100];` ⟶ 2 samples backward (earlier) in buffer?

Modulo arithmetic

- `x % y` ("x mod y") gives the remainder after `x` is divided by `y`
  - If `x > 0` and `y > 0`, then the range of `x % y` is 0 to y-1
    - For example: 5 % 2 = 1
  - But if `x < 0`, result will be negative: -(y-1) to 0
    - For example: -5 % 2 = -1
    - This is clearly not what we want!
    - Even worse, it is language-dependent. This is not, for example, not how Python implements modulo.
- What is the solution to keep the indices in range?
  - Always add one or more multiples of the buffer size

```
std::vector<float> gDelayBuffer;
float twoSamplesBeforeN =
          gDelayBuffer[(n - 2 + gDelayBuffer.size()) % gDelayBuffer.size()];
```

  - Here, even if `n < 2`, the modulo will be positive

Circular buffer task 2

- Task: without changing buffer size, change delay to 0.1 seconds
  - Now you can no longer read the oldest sample in the buffer
  - You will need to use modulo arithmetic on the write pointer to look backward by 0.1 seconds
  - How many samples is 0.1 seconds at 44.1kHz sample rate?

Circular buffer code

```
std::vector<float> gDelayBuffer;
unsigned int gWritePointer = 0;
unsigned int gOffset = 0;

bool setup(BelaContext *context, void *userData)
{
    // Allocate the circular buffer to 0.5 seconds
    gDelayBuffer.resize(0.5 * context->audioSampleRate);
    // Calculate the offset based on the sample rate
    gOffset = 0.1*context->audioSampleRate;
    return true;
}

void render(BelaContext *context, void *userData)
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        float in = 

        // Read the output from the write pointer (oldest sample)
        float out = gDelayBuffer[(gWritePointer - gOffset + gDelayBuffer.size()) % gDelayBuffer.size()];

        // Overwrite the buffer at the write pointer, then increment and wrap pointer
        gDelayBuffer[gWritePointer] = in;
        gWritePointer++;
        if(gWritePointer >= gDelayBuffer.size())
            gWritePointer = 0;
        // [...]
    }
}
```

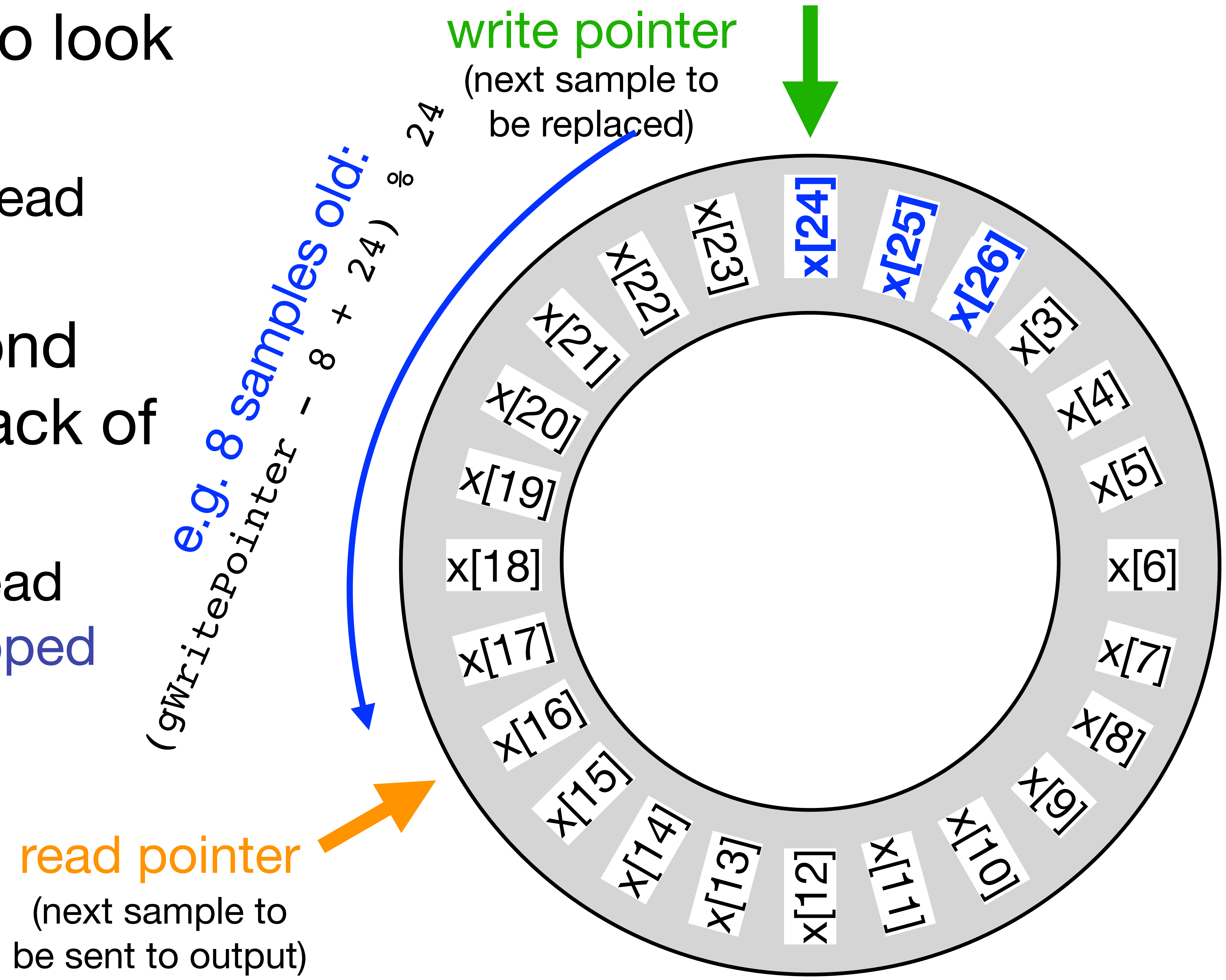
offset between pointers in samples

offset calculated here

modulo calculation

Write and read pointers

- We can use modulo arithmetic to look backwards in the circular buffer
  - We could do this every sample to read samples out at a particular delay
- Alternatively, we can use a second pointer (read pointer) to keep track of where we are reading
  - Each sample, both the write and read pointers are incremented and wrapped
  - The distance between the pointers determines the delay



Write and read pointers

- The circular buffer now has **three** components:
  1. A region in memory (array) to store the samples
  2. A write pointer to keep track of where we are writing new samples
  3. A read pointer to keep track of where we are reading old samples
- Remember, there is (still) no functional beginning or end to a circular buffer!
- In code, we need to declare three (global) variables:

```
std::vector<float> gDelayBuffer;
unsigned int gWritePointer = 0;
unsigned int gReadPointer =
```

What index we put here determines the delay

  - For each new sample, store it at the write pointer, then increment/wrap both pointers

```
out = gDelayBuffer[gReadPointer];
gReadPointer++;
if(gReadPointer >= gDelayBuffer.size())
    gReadPointer = 0;
```

```
gDelayBuffer[gWritePointer] = in;
gWritePointer++;
if(gWritePointer >= gDelayBuffer.size())
    gWritePointer = 0;
```

Circular buffer task 3

- Task: Change your code to implement the delay using a read pointer
  - Keep the delay at 0.1 seconds
  - The delay is set up by the difference between the read and write pointer locations
  - You should not need modulo indexing in `render()` anymore

```
std::vector<float> gDelayBuffer;
unsigned int gWritePointer = 0;
unsigned int gReadPointer =
```

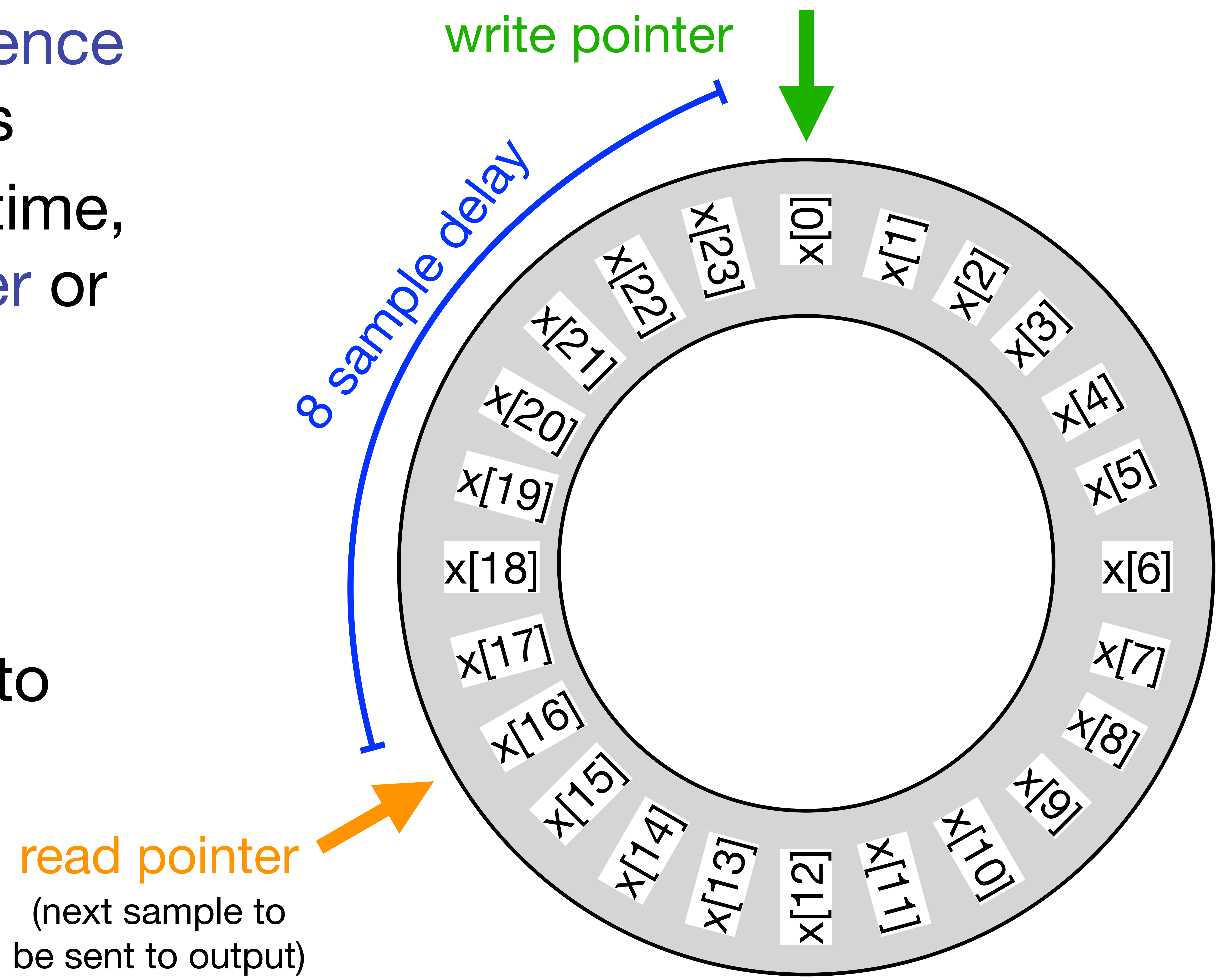
What index we put here determines the delay

```
out = gDelayBuffer[gReadPointer];
gReadPointer++;
if(gReadPointer >= gDelayBuffer.size())
    gReadPointer = 0;
```

```
gDelayBuffer[gWritePointer] = in;
gWritePointer++;
if(gWritePointer >= gDelayBuffer.size())
    gWritePointer = 0;
```

Adjusting the delay

- Delay time is given by the difference between read and write pointers
- If we want to change the delay time, should we move the read pointer or the write pointer?
  - The read pointer. Why?
  - Don't want a gap in the buffer
- When the delay length should change, use modulo arithmetic to recalculate the read pointer

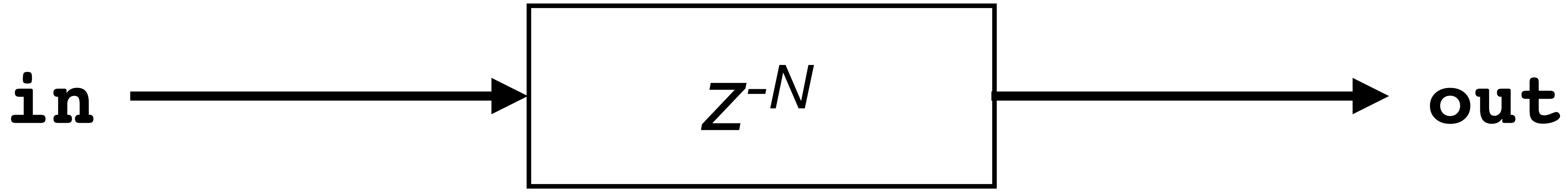


Circular buffer task 4

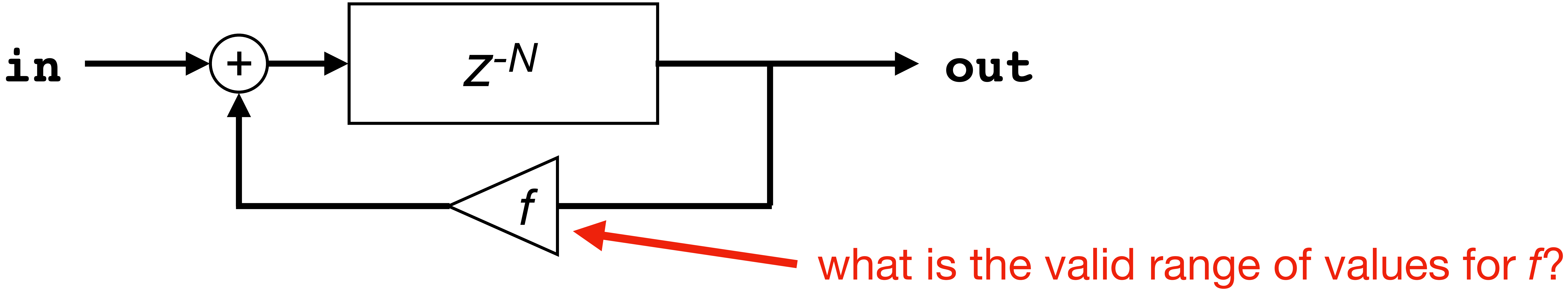
- Task: add a GUI slider to change the delay length
  - See Lecture 4 for more details on the Bela GUI
  - Make the delay adjustable between 0 and 0.49 seconds
  - From time in seconds, calculate how many samples of delay are needed
  - Update the location of the read pointer based on the write pointer location
- Hint: make sure your code works with a delay of 0!
  - It might matter whether you read or write to the buffer first

Echo effect

- With our circular buffer, we have implemented a simple delay:



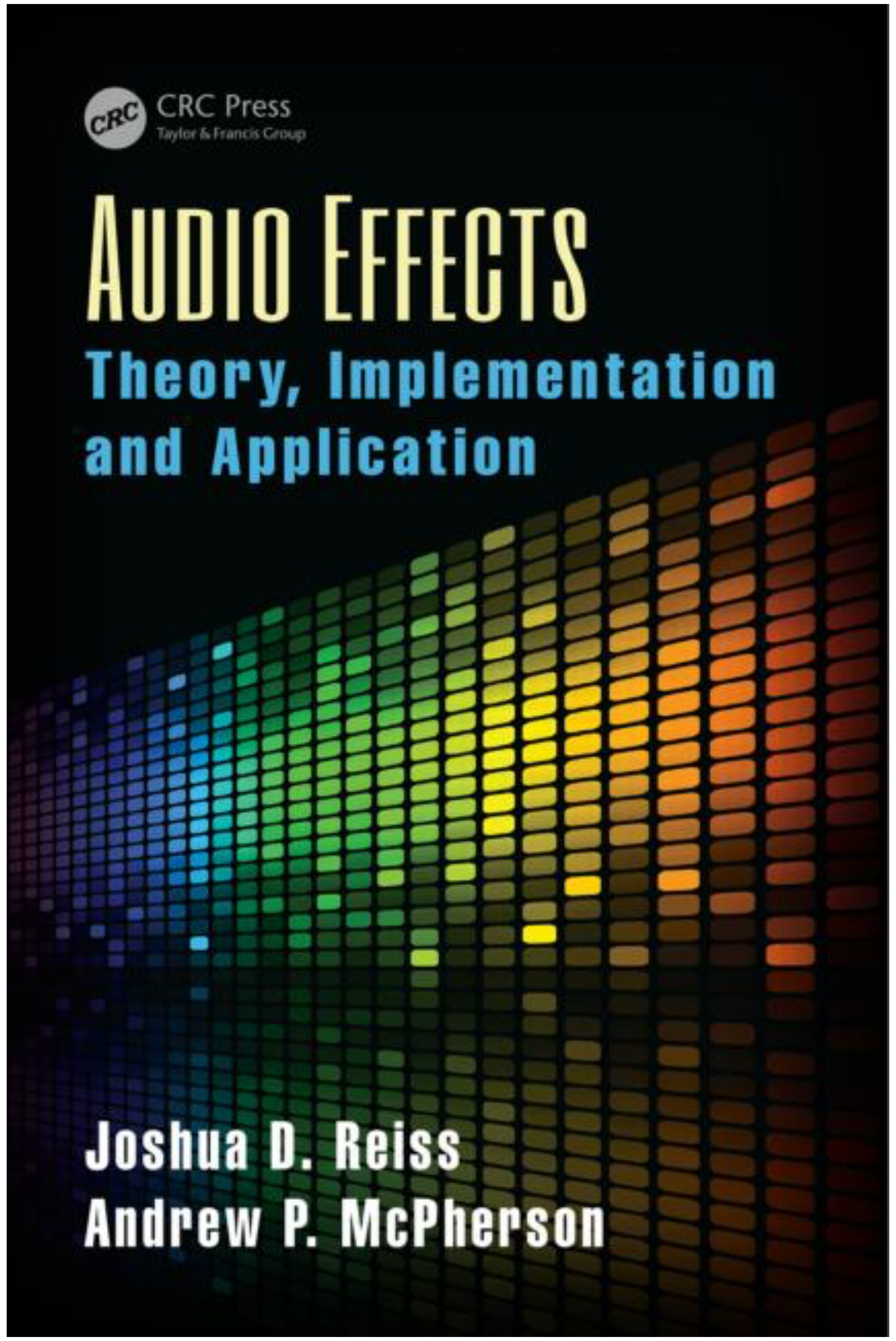
- We can also add feedback (or regeneration) from output to input
  - This produces periodic echoes of the sound



- Task: add feedback to the circular-buffer project to create an echo
  - Add a second GUI slider to control the level of feedback

Other delay-based effects

- Vibrato
  - Created with a variable-speed read pointer
- Chorus
  - A time-varying delayed copy added to original signal
- Flanger
  - Implemented like a chorus, but with lower delay and possible presence of feedback
- All of these require fractional read pointers (see Lecture 3)
- They also require LFOs (low-frequency oscillators)
- The Audio Effects textbook has more theory and code examples for these effects



Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources