# Effects App

# Motivation

- Need to publish app using Oboe for Marmot identification
- Simultaneous input/output is a common difficult use case of low latency mode of Oboe
- Crash reports from devices using Oboe in the field

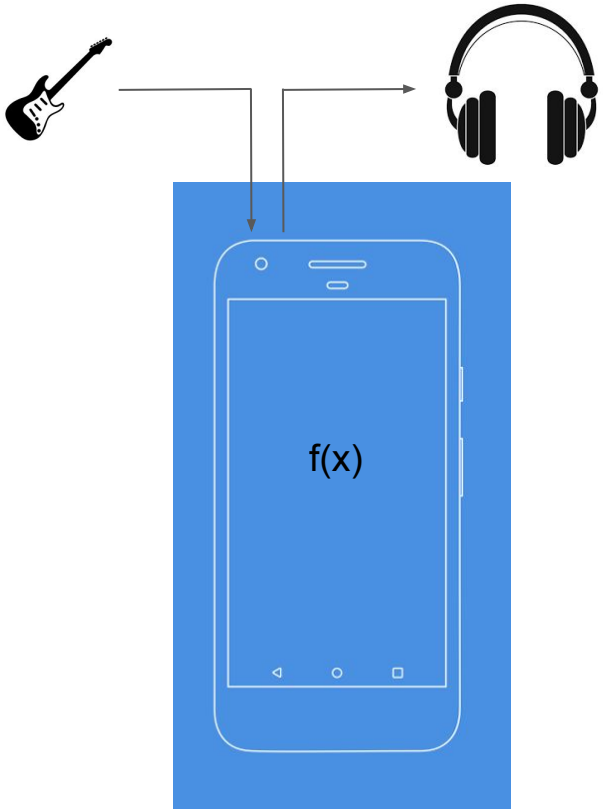
# Design Goals

- Effects which are..
  - Portable - minimum contract/overhead
  - Versatile - formats, sample rates
  - Integrable - add new effects easily
  - Performant - mem/cpu
- Oboe processing engine is extensible/re-usable

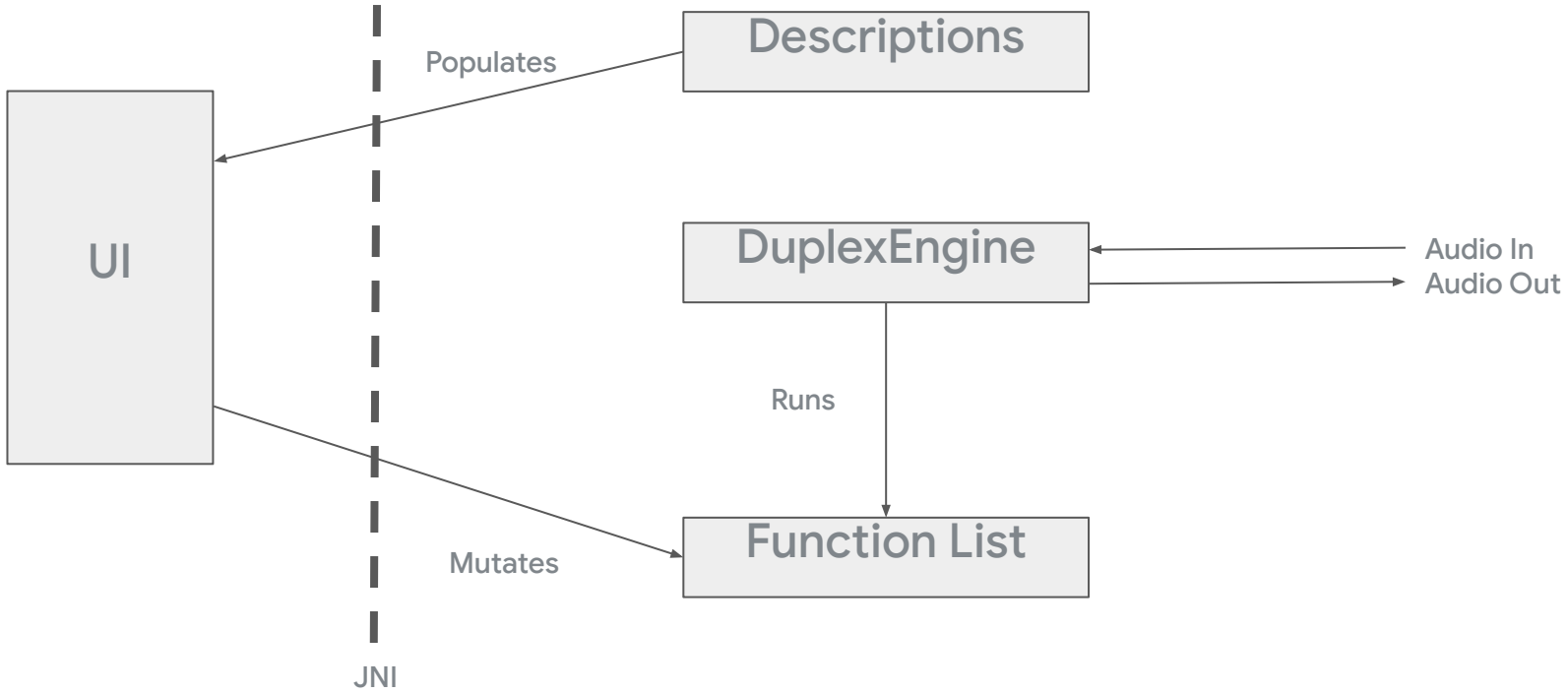
# UI/UX Goals

- Intuitive/Interactive controls
- Implements best design practices (Material)

# How to Use

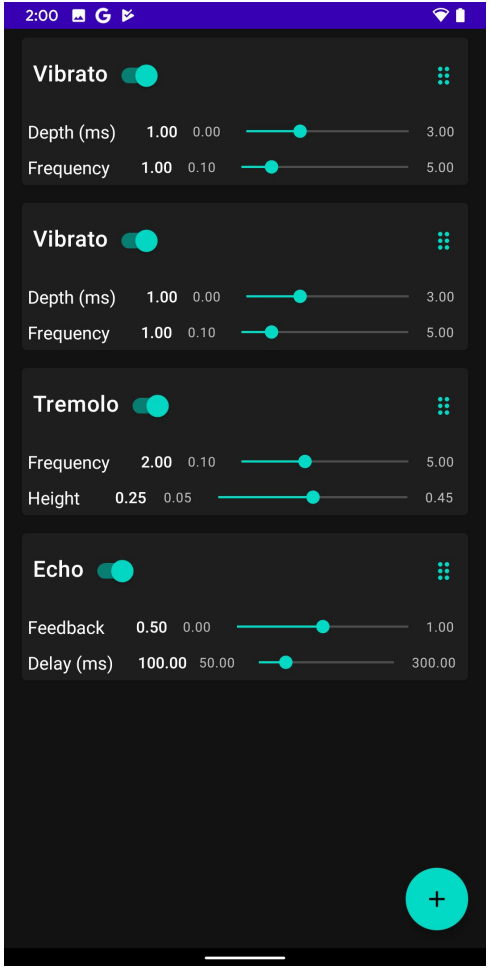


# Architecture

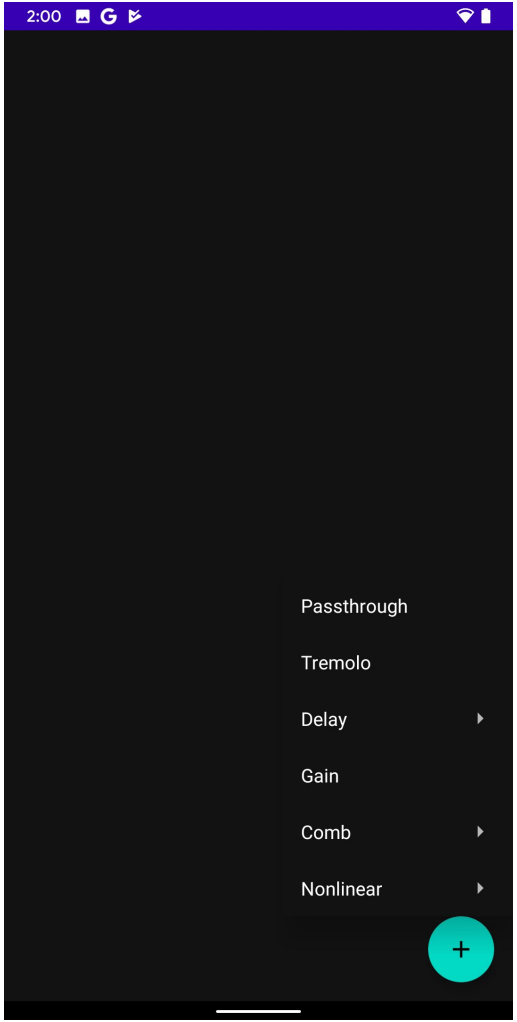


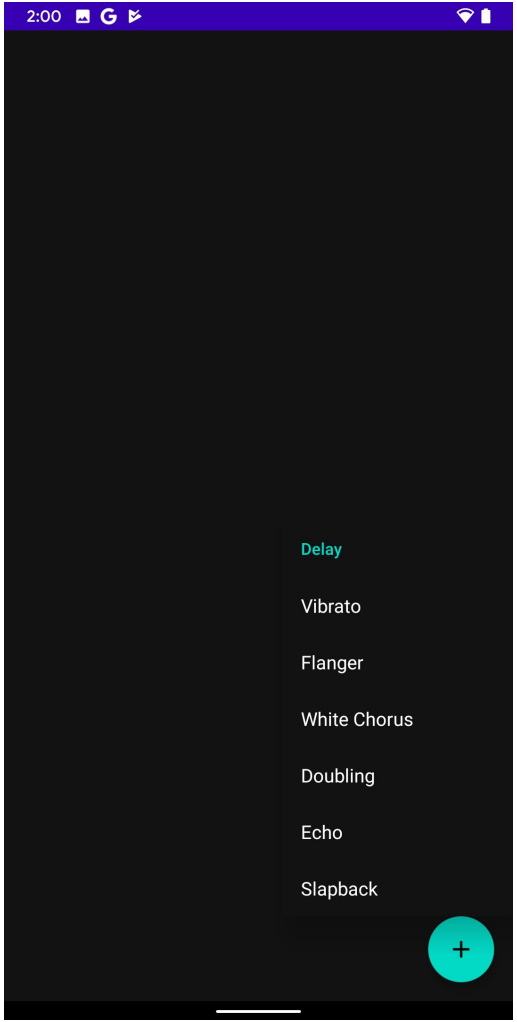
# UI

# Layout

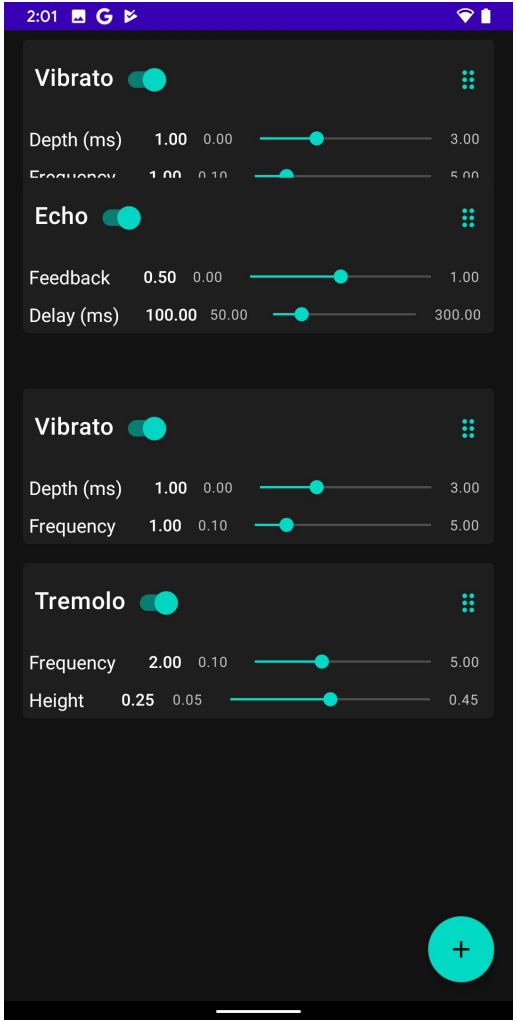

2:00
Vibrato
Depth (ms) 1.00 0.00 3.00
Frequency 1.00 0.10 5.00
Vibrato
Depth (ms) 1.00 0.00 3.00
Frequency 1.00 0.10 5.00
Tremolo
Frequency 2.00 0.10 5.00
Height 0.25 0.05 0.45
Echo
Feedback 0.50 0.00 1.00
Delay (ms) 100.00 50.00 300.00
+

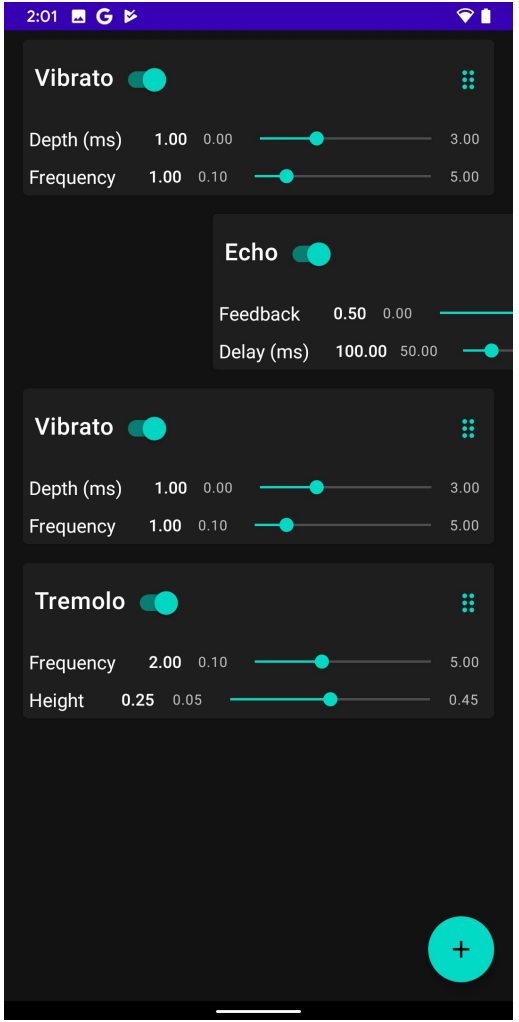
# Adding an Effect



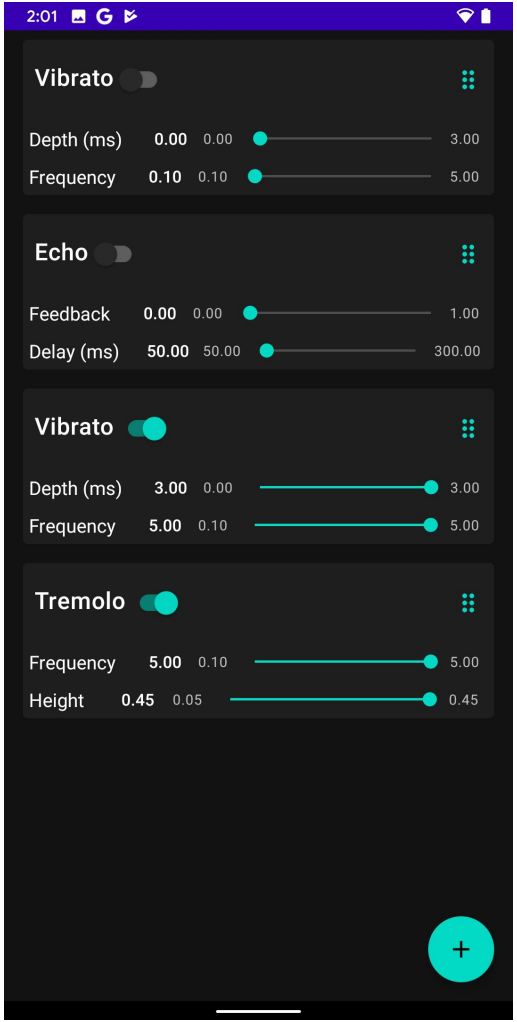



# Reordering and Removing



# Modifying Parameters

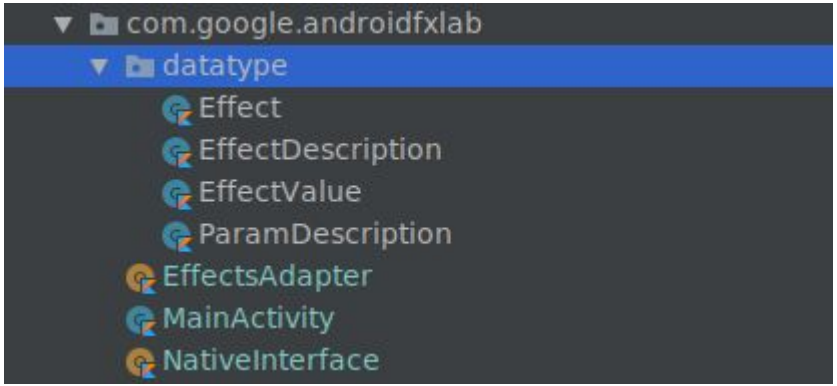


# Effects Framework

# Big Ideas

- How do we describe effects to the UI?
- Heterogeneous collections of effects
- Allow effects to operate on anything*
- Warning -- lots of code screenshots

# Kotlin side




```kotlin
// State of audio engine
external fun createAudioEngine()

external fun destroyAudioEngine()

// These functions populate effectDescriptionMap
private external fun getEffects(): Array<EffectDescription>

// These functions mutate the function list
// Adds effect at index
private external fun addDefaultEffectNative(id: Int)

private external fun removeEffectNative(index: Int)

private external fun rotateEffectNative(from: Int, to: Int)

private external fun modifyEffectNative(id: Int, index: Int, params: FloatArray)

private external fun enableEffectNative(index: Int, enable: Boolean)

private external fun enablePassthroughNative(enable: Boolean)
```

```kotlin
// Used to load the 'native-lib' library on application startup.
val effectDescriptionMap: Map<String, EffectDescription>

init {
    System.loadLibrary( libname: "native-lib")
    effectDescriptionMap = getEffects().associateBy { it.name }
    Log.d( tag: "MAP", effectDescriptionMap.toString())
}
```

# Descriptions

- Compile-time knowledge
- Easy to add
- _ef = std::function<void(iter_type, iter_type)>

```
// EffectType is the description subclass, N is num of params
// Function implementations in this class contain shared behavior
// Which can be shadowed.
    <EffectType,size_t N>
    class EffectDescription {
    public:
        // These methods will be shadowed by subclasses

        static constexpr size_t getNumParams() {
            return N;
        }

        static constexpr std::array<float, N> getEmptyParams() {
            return std::array<float, EffectType::getNumParams()>();
        }

        static constexpr std::string_view getName();

        static constexpr std::string_view getCategory();

        static constexpr std::array<ParamType, N> getParams();

        <iter_type>
        static _ef<iter_type> buildEffect(std::array<float, N> paramArr);


        <iter_type>
        static _ef<iter_type> buildDefaultEffect() {...}

        // The default behavior is new effect, can be shadowed
        <iter_type>
        static _ef<iter_type> modifyEffect(
                _ef<iter_type> /* effect */, std::array<float, N> paramArr) {...}
        <iter_type>
        static _ef<iter_type> modifyEffectVec(
                _ef<iter_type> effect, std::vector<float> paramVec) {...}
    };

} // namespace effect
```

# Adding an Effect

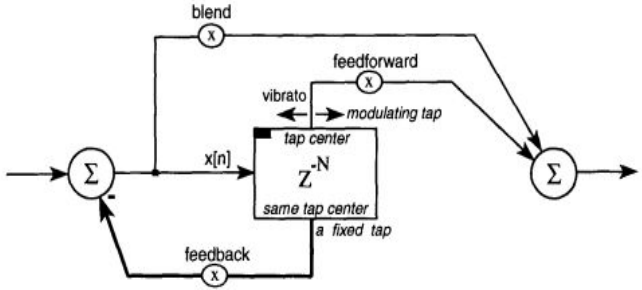
```
namespace Effect {

class EchoDescription: public EffectDescription<EchoDescription, 2> {
public:
    static constexpr std::string_view getName() {
        return std::string_view("Echo");
    }

    static constexpr std::string_view getCategory() {
        return std::string_view("Delay");
    }

    static constexpr std::array<ParamType, getNumParams()> getParams() {
        return std::array<ParamType, getNumParams()> {
                ParamType("Feedback", 0, 1, 0.5),
                ParamType("Delay (ms)", 50, 300, 100),
        };
    }
    <iter_type>
    static _ef<iter_type> buildEffect(std::array<float, getNumParams()> paramArr) {
        return _ef<iter_type> {
                EchoEffect<iter_type>{paramArr[0], paramArr[1]}
        };
    }
};

} //namespace Effect
```

# One Source for Effects

```
constexpr std::tuple<
        Effect::PassthroughDescription,
        Effect::TremoloDescription,
        Effect::VibratoDescription,
        Effect::GainDescription,
        Effect::FlangerDescription,
        Effect::WhiteChorusDescription,
        Effect::FIRDescription,
        Effect::IIRDescription,
        Effect::AllPassDescription,
        Effect::DoublingDescription,
        Effect::OverdriveDescription,
        Effect::DistortionDescription,
        Effect::EchoDescription,
        Effect::SlapbackDescription
> EffectsTuple{};

constexpr size_t numEffects = std::tuple_size<decltype(EffectsTuple)>::value;
```

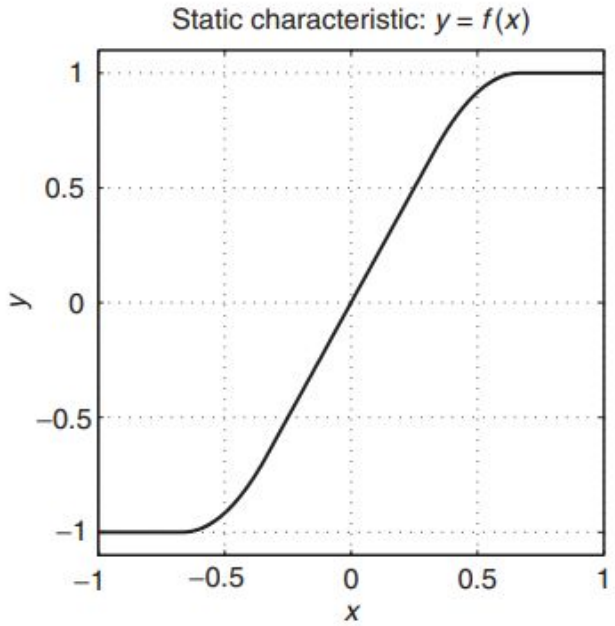
# Some Effects (class)

```cpp
template <class iter_type>
class DelayLineEffect {

public:
    // delay > 0, depth in samples, mod is control signal
    DelayLineEffect(float blend, float feedForward, float feedBack, int delay, int depth, std::function<float()> &&mod) :
        kBlend(blend),
        kFeedForward(feedForward),
        kFeedBack(feedBack),
        kDelay(delay),
        kDepth(depth),
        mMod(mod) { }


    void operator () (typename std::iterator_traits<iter_type>::reference x) {
        auto delayInput = x + kFeedBack * delayLine[kTap];
        auto variableDelay = mMod() * kDepth + kTap;
        int index = static_cast<int>(variableDelay);
        auto fracComp = 1 - (variableDelay - index);
        //linear
        // auto interpolated = fracComp * delayLine[index] + (1 - fracComp) * delayLine[index + 1];
        // all - pass
        float interpolated = fracComp * delayLine[index] + delayLine[index + 1]
                - fracComp * prevInterpolated;

        prevInterpolated = interpolated;
        delayLine.push(delayInput);
        x = interpolated * kFeedForward + kBlend * delayInput;
    }
    void operator () (iter_type begin, iter_type end) {
        for (; begin != end; ++begin) {
            operator()(*begin);
        }
    }
```



# Some Effects (function)

```
<floating>
void _overdrive (floating &x) {
    static constexpr double third = (1.0 / 3.0);
    auto abs = std::abs(x);
    if (abs <= third) {
        x *= 2;
    } else if (abs <= 2 * third) {
        x = std::copysign((3 - (2 - 3 * abs) * (2 - 3 * abs)) * third, x);
    } else {
        x = std::copysign(1, x);
    }
}

<iter_type>
void overdrive(iter_type beg, iter_type end) {
    for (; beg != end; ++beg){
        _overdrive(*beg);
    }
}
```



# FunctionList -- putting it all together

```
<iter_type>
class FunctionList {
    std::vector<std::pair<std::function<void(iter_type, iter_type)>, bool>> functionList;
    bool muted = false;
public:
    FunctionList() = default;

    FunctionList(const FunctionList &) = delete;

    FunctionList &operator=(const FunctionList &) = delete;


    void operator()(iter_type begin, iter_type end) {
        for (auto &f : functionList) {
            if (f.second == true) std::get<0>(f)(begin, end);
        }
        if (muted) std::fill(begin, end, 0);
    }
```

# std::function and type erasure

- The benefits of polymorphism (collections, flexible functions) without
  - Virtual Lookup
  - Inheritance contracts
  - Classes
  - Runtime danger

# Flexibility of iterators -- <class iter_type>

- Operate on arbitrary type
  - Float/double, int16/32?
- Operate on arbitrary data source
  - Arbitrary range (single sample, buffers of [size])
  - Containers (not raw pointers)
  - File iterators

# Duplex Engine

```
class DuplexEngine {
public:
    DuplexEngine();

    void beginStreams();

    virtual ~DuplexEngine() = default;

    oboe::Result startStreams();

    oboe::Result stopStreams();


    std::variant<FunctionList<int16_t *>, FunctionList<float *>> functionStack{
            std::in_place_type<FunctionList<int16_t *>>};

private:

    void openInStream();

    void openOutStream();

    static oboe::AudioStreamBuilder defaultBuilder();

    <numeric>
    void createCallback();

    oboe::ManagedStream inStream;
    std::unique_ptr<oboe::AudioStreamCallback> mCallback;
    oboe::ManagedStream outStream;


};
```

# Future Work

- Fixed point overflow
- Recording and other UI extras
- Many more effects!
- Publishing/Open-sourcing

# Live Demo