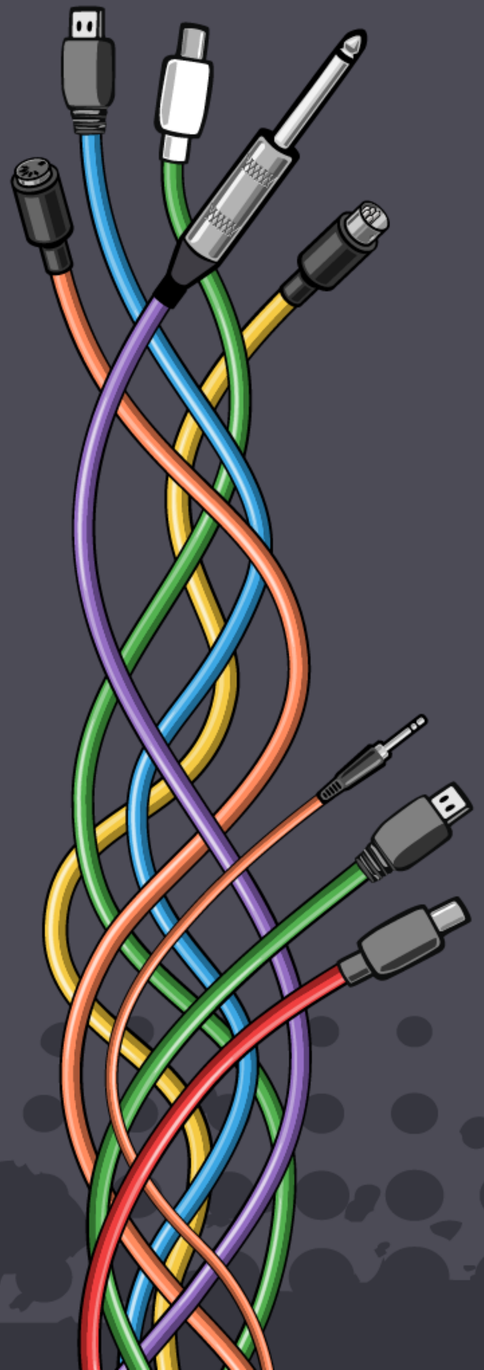

ENUMERATE AND EXTRACT AUDIO BUFFERS WHEN DEBUGGING C++ APPLICATIONS

HENNING MEYER

ADC25 Bristol

Debugging Applications

Enterprise Applications:

- Enterprise applications are concerned with crashes and security vulnerabilities due to out of bounds memory accesses.
- Security vulnerabilities have become big business and a lot of new tooling focuses on finding or exploiting vulnerabilities.
- Defects in applications that are not security problems are not treated as high priority.

Debugging Applications

Enterprise Applications:

- Enterprise applications are concerned with crashes and security vulnerabilities due to out of bounds memory accesses.
- Security vulnerabilities have become big business and a lot of new tooling focuses on finding or exploiting vulnerabilities.
- Defects in applications that are not security problems are not treated as high priority.

Signal Processing Applications:

- Signal processing algorithms may require complex addressing schemes that can be 100% memory safe and still be wrong.
- Signal processing bugs will ruin the experience and cannot be ignored.
- We cannot tell whether data is valid by looking at a single sample, we need to evaluate the entire buffer.
- Standard debugging tools have poor support for that.
- Lots of custom tooling in various code bases.

Turning Unstructured Data into Structured Data

- Memory Contents in a C++ application are structured via the type system.
- Debuggers are unaware of this structure except for local and global variables.
- C++ applications use RAII to manage memory and other resources.
- Typical examples are containers like `std::vector` and `std::unordered_map` and smart pointers like `std::shared_ptr` and `std::unique_ptr`.
- Production code bases have many examples of containers and smart pointers outside the standard library.
- Ultimately all resources are owned by local and global variables either directly or indirectly.

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

(gdb)

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb)
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb)
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
$3 = 0x40020
(gdb)
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
$3 = 0x40020
(gdb) x 0x40020
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
$3 = 0x40020
(gdb) x 0x40020
0x40020 <data>: 0x500b0
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
$3 = 0x40020
(gdb) x 0x40020
0x40020 <data>: 0x500b0
(gdb) x 0x500b0
```

Examing Memory with GDB

Code

```
struct Data {
  int val = 42;
  std::vector<int> more =
    {1, 2, 3};
};

std::unique_ptr<Data> data =
  std::make_unique<Data>();
```

GDB commands

```
(gdb) print data
$1 = {get() = 0x500b0}
(gdb) print *data
$2 = {val=42, more={1, 2, 3}}
(gdb) print &data
$3 = 0x40020
(gdb) x 0x40020
0x40020 <data>: 0x500b0
(gdb) x 0x500b0
0x500b0: 0x0000002a
```

Examing Memory with GDB

- GDB can work forwards from a container or smart pointer to the addresses of the contained elements
- it cannot work backwards from the address to "I know there is a `struct Data` at this address"
- it only supports this if someone has manually created a pretty printer for that type

Examing Memory with GDB

- GDB can work forwards from a container or smart pointer to the addresses of the contained elements
- it cannot work backwards from the address to "I know there is a `struct Data` at this address"
- it only supports this if someone has manually created a pretty printer for that type
- I want to support all containers and smart pointers and I don't want to write pretty printers by hand

Examing Memory with GDB

- GDB can work forwards from a container or smart pointer to the addresses of the contained elements
- it cannot work backwards from the address to "I know there is a `struct Data` at this address"
- it only supports this if someone has manually created a pretty printer for that type
- I want to support all containers and smart pointers and I don't want to write pretty printers by hand
- I want to traverse all containers, store the addresses of found objects and allow reverse lookup by address

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

Question

How can a C++ compiler iterate over elements in a container?

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

```
(gdb) print container
```

Question

How can a C++ compiler iterate over elements in a container?

```
for(auto&& e : container) {
  std::cout « e « std::endl;
}
```

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

```
(gdb) print container
```

Question

How can a C++ compiler iterate over elements in a container?

```
for(auto&& e : container) {
  std::cout « e « std::endl;
}
```

Answer

By calling `begin()` and `end()` behind the scenes.

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

```
(gdb) print container
```

Question

How can a C++ compiler iterate over elements in a container?

```
for(auto&& e : container) {
  std::cout « e « std::endl;
}
```

Answer

By calling `begin()` and `end()` behind the scenes.
By calling `operator*()` and `operator++()` on the obtained iterators.

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

```
(gdb) print container
```

Answer

By executing `begin()` and `end()` behind the scenes.

Question

How can a C++ compiler iterate over elements in a container?

```
for(auto&& e : container) {
  std::cout « e « std::endl;
}
```

Answer

By calling `begin()` and `end()` behind the scenes.
By calling `operator*()` and `operator++()` on the obtained iterators.

Generically Iterate over any Container

Question

How can a C++ debugger iterate over elements in a container?

```
(gdb) print container
```

Answer

By executing `begin()` and `end()` behind the scenes.
By executing `operator*()` and `operator++()` on the obtained iterators.

Question

How can a C++ compiler iterate over elements in a container?

```
for(auto&& e : container) {
  std::cout « e « std::endl;
}
```

Answer

By calling `begin()` and `end()` behind the scenes.
By calling `operator*()` and `operator++()` on the obtained iterators.

Object Enumeration Algorithm

1. find all containers, start with global and local variables
2. run an interpreter on the machine code of `begin()` and `end()` to get iterators
3. run an interpreter on the machine code of `operator*()` to get a valid pointer
4. run an interpreter on the machine code of `operator++()` to get more iterators

Object Enumeration Algorithm

1. find all containers, start with global and local variables
2. run an interpreter on the machine code of `begin()` and `end()` to get iterators
3. run an interpreter on the machine code of `operator*()` to get a valid pointer
4. run an interpreter on the machine code of `operator++()` to get more iterators
5. This relies on the public interface of a type and works for user-defined container types without requiring additional configuration

Interpreter for Machine Code

Here is the complete disassembly for the necessary functions in a libstdc++ container on x86-64:

std::string::begin()

```
endbr64
movq (%rdi), %rax
retq
```

std::string::end()

```
endbr64
movq (%rdi), %rax
addq 8(%rdi), %rax
retq
```

Object Enumeration beyond Containers

1. Smart pointers work just the same: use `operator bool()` and `operator*()`.
2. Smart pointers might be declared with an interface type, not the concrete type. The concrete type can be recovered if the declared type has virtual methods.
3. `optional` and `expected` work just the same: `has_value()` and `value()`.
4. `std::function` does not have a public interface to get the contained value.
5. `std::variant` will be discussed later.

Object Enumeration: Theory vs Practice

Let us assume the method works flawlessly:

- We will discover millions of objects, thousands of types
- It is not enough to make the debugger aware of objects, we need to find ways to make use of this information without causing information overload.
- I want to allow debugging in terms of the application's object model, not raw bits and bytes

Object Enumeration: Theory vs Practice

Let us assume the method works flawlessly:

- We will discover millions of objects, thousands of types
- It is not enough to make the debugger aware of objects, we need to find ways to make use of this information without causing information overload.
- I want to allow debugging in terms of the application's object model, not raw bits and bytes

Can we assume the method works flawlessly?

- Release builds often show `optimized` away for local variables.
- Release builds inline simple functions and remove unused functions.

Object Enumeration: Release Builds

Can we assume the method works flawlessly?

- Release builds often show `optimized` away for local variables.
- Release builds inline simple functions and remove unused functions.

Object Enumeration: Release Builds

Can we assume the method works flawlessly?

- Release builds often show `optimized` away for local variables.
- Release builds inline simple functions and remove unused functions.

A variable will show as `optimized` away if the debugger cannot get its memory location.

- Variables with a non-trivial destructor typically reside in memory and not in registers. A pointer, index or iterator will be optimized more aggressively than a container or smart pointer.

Object Enumeration: Release Builds

Can we assume the method works flawlessly?

- Release builds often show `optimized away` for local variables.
- Release builds inline simple functions and remove unused functions.

A variable will show as `optimized away` if the debugger cannot get its memory location.

- Variables with a non-trivial destructor typically reside in memory and not in registers. A pointer, index or iterator will be optimized more aggressively than a container or smart pointer.
- Missing functions are quite different from missing variables. Functions don't have state.

Object Enumeration: Release Builds

Can we assume the method works flawlessly?

- Release builds often show `optimized away` for local variables.
- Release builds inline simple functions and remove unused functions.

A variable will show as `optimized away` if the debugger cannot get its memory location.

- Variables with a non-trivial destructor typically reside in memory and not in registers. A pointer, index or iterator will be optimized more aggressively than a container or smart pointer.
- Missing functions are quite different from missing variables. Functions don't have state.
- Functions can be recovered from an unoptimized build and used on the optimized build (we run them in an interpreter anyway).

Object Enumeration: Build Recommendations

- All builds, even Release builds, should be built with debugging information. You can strip the debugging information before shipping code to customers.
- Debug information should be as detailed as possible (use the compiler option `-g3`, not just `-g`). You can use compressed debug information if the size of debug information is a concern.
- There should be an unoptimized build that is identical to the release build except for optimization flags. Sometimes Debug builds contain additional code or even additional members in `structs` compared to Release builds.
- Use the most recent compiler available.

Visualizing Memory Contents

- Uncompressed images are easiest. We know how to show a simplified image: by downscaling
- There is still a lot of freedom in encoding the pixels in images
- Uncompressed audio is a lot more straightforward.
- Samples will be encoded as signed integers or floating point numbers.
- Unlike images, there is a direct correspondence between C++ built-in types and sample encodings
- 24bit integers have been added to C23 but not yet to C++ (clang supports them in C++ as an extension)
- metadata consists of the sample rate and number of channels

Extracting Samples and Metadata

Application 1

Hard-coded sample rate, hard-coded numer of channels, interleaved stereo

Application 2

using `juice::AudioBuffer<T>`, templated by sample type and non-interleaved channels

Application 3

an application that uses one type that can hold different sample types at runtime, specified via an enum

Extracting Samples and Metadata

Application 1

Hard-coded sample rate, hard-coded numer of channels, interleaved stereo

Application 2

using `juice::AudioBuffer<T>`, templated by sample type and non-interleaved channels

Application 3

an application that uses one type that can hold different sample types at runtime, specified via an enum

There is no uniform interface, there is no std::audio concept in C++.

Extracting Samples and Metadata

Application 1

Hard-coded sample rate, hard-coded numer of channels, interleaved stereo

Application 2

using `juice::AudioBuffer<T>`, templated by sample type and non-interleaved channels

Application 3

an application that uses one type that can hold different sample types at runtime, specified via an enum

There is no uniform interface, there is no std::audio concept in C++. This will require some kind of configuration for the debugger:

- XML like NatVis used in Microsoft products
- Python code as used by GDB and LLDB
- configuration via functions in the binary used by the debugger

Extracting Samples and Metadata

Application 1

Hard-coded sample rate, hard-coded numer of channels, interleaved stereo

Application 2

using `juce::AudioBuffer<T>`, templated by sample type and non-interleaved channels

Application 3

an application that uses one type that can hold different sample types at runtime, specified via an enum

There is no uniform interface, there is no std::audio concept in C++.
This will require some kind of configuration for the debugger:

- XML like NatVis used in Microsoft products
- Python code as used by GDB and LLDB
- configuration via functions in the binary used by the debugger
  - `get_sample_rate()`
  - `get_number_of_channels()`
  - `get_number_of_samples()`
  - either `get_interleaved_samples()` or `get_samples_for_channel()` or both

Connecting samples with metadata

How can we use a C++ function to tell us at runtime about the compile-time type used for the samples in a buffer?

`std::variant`

- each object may hold a different type at runtime
- the `get()` free function can be used to get a reference to the contents

`V = std::variant<int,float>` can be queried with

- `int& std::get<int>(V&)` and
- `float& std::get<float>(V&)`

Exactly one of these functions will return a reference to valid data, the others will not return a value. The succeeding function allows us to infer the type.

A C++ object for holding samples and metadata

- We use templated free functions to get a pointer to samples.

Connecting samples with metadata

How can we use a C++ function to tell us at runtime about the compile-time type used for the samples in a buffer?

`std::variant`

- each object may hold a different type at runtime
- the `get()` free function can be used to get a reference to the contents

`V = std::variant<int,float>` can be queried with

- `int& std::get<int>(V&)` and
- `float& std::get<float>(V&)`

Exactly one of these functions will return a reference to valid data, the others will not return a value. The succeeding function allows us to infer the type.

A C++ object for holding samples and metadata

- We use templated free functions to get a pointer to samples.
- If it is not the right type the function can throw, `abort()` or return `nullptr`.

Connecting samples with metadata

How can we use a C++ function to tell us at runtime about the compile-time type used for the samples in a buffer?

`std::variant`

- each object may hold a different type at runtime
- the `get()` free function can be used to get a reference to the contents

`V = std::variant<int,float>` can be queried with

- `int& std::get<int>(V&)` and
- `float& std::get<float>(V&)`

Exactly one of these functions will return a reference to valid data, the others will not return a value. The succeeding function allows us to infer the type.

A C++ object for holding samples and metadata

- We use templated free functions to get a pointer to samples.
- If it is not the right type the function can `throw`, `abort()` or return `nullptr`.
- We can infer the type from the return type of the succeeding function.

Connecting samples with metadata

How can we use a C++ function to tell us at runtime about the compile-time type used for the samples in a buffer?

`std::variant`

- each object may hold a different type at runtime
- the `get()` free function can be used to get a reference to the contents

`V = std::variant<int,float>` can be queried with

- `int& std::get<int>(V&)` and
- `float& std::get<float>(V&)`

Exactly one of these functions will return a reference to valid data, the others will not return a value. The succeeding function allows us to infer the type.

A C++ object for holding samples and metadata

- We use templated free functions to get a pointer to samples.
- If it is not the right type the function can `throw`, `abort()` or return `nullptr`.
- We can infer the type from the return type of the succeeding function.
- There may be multiple channels. The samples of different channels may or may not be interleaved.

Connecting samples with metadata

How can we use a C++ function to tell us at runtime about the compile-time type used for the samples in a buffer?

`std::variant`

- each object may hold a different type at runtime
- the `get()` free function can be used to get a reference to the contents

`V = std::variant<int,float>` can be queried with

- `int& std::get<int>(V&)` and
- `float& std::get<float>(V&)`

Exactly one of these functions will return a reference to valid data, the others will not return a value. The succeeding function allows us to infer the type.

A C++ object for holding samples and metadata

- We use templated free functions to get a pointer to samples.
- If it is not the right type the function can `throw`, `abort()` or return `nullptr`.
- We can infer the type from the return type of the succeeding function.
- There may be multiple channels. The samples of different channels may or may not be interleaved.
- We use one free function to query for interleaved samples and a different one for non-interleaved samples

Connecting samples with metadata

Example for a type supporting only 44.1kHz stereo 16bit PCM

```
unsigned get_sample_rate(const CdAudioBuffer&) { return 44100;}
unsigned short get_number_of_channels(const CdAudioBuffer&) { return 2;}
unsigned get_number_of_samples(const CdAudioBuffer&);
const int16_t* get_interleaved_samples(const CdAudioBuffer&);
```

Example for a type in intermediate processing

```
double get_sample_rate(const ResampledBuffer&);
unsigned get_number_of_channels(const ResampledBuffer&);
size_t get_number_of_samples(const ResampledBuffer&);
std::span<const float> get_samples_from_channel(const ResampledBuffer&, unsigned c);
```

Connecting samples with metadata

Example for a templated type

```
template<typename T>
double get_sample_rate(const juce::AudioBuffer<T>&) {
return 0.0; }
template<typename T>
unsigned get_number_of_channels(const juce::AudioBuffer<T>& b) {
return b.getNumChannels(); }
template<typename T>
size_t get_number_of_samples(const juce::AudioBuffer<T>& b) {
return b.getNumSamples(); }
template<typename T>
const T* get_samples_for_channel(const juce::AudioBuffer<T>& b, int c) {
return b.getReadPointer(c); }
```

Connecting samples with metadata

Example for a type supporting multiple encodings at runtime

```
unsigned get_sample_rate(const SampleBuffer&);
unsigned short get_number_of_channels(const SampleBuffer&);
size_t get_number_of_samples(const SampleBuffer&);
template <typename T>
const T* get_interleaved_samples(const SampleBuffer&);
template <>
const int16_t* get_interleaved_samples<int16_t>(const SampleBuffer&);
template <>
const float* get_interleaved_samples<float>(const SampleBuffer&);
template <typename T>
const T* get_samples_from_channel(const SampleBuffer&, unsigned );
template <>
const int16_t* get_interleaved_samples(const SampleBuffer&, unsigned );
```

Connecting samples with metadata

The advantage of using free functions is that we can add them after the fact without modifiying the source or requiring a rebuild of the main application:

- Add a new unit test that implements these functions.
- Add the test binary as context to the debugger when debugging the main application.

Detected Audio Data

1. We can create a list of all objects in memory
2. We can specify which types hold audio data and how to get metadata for them.
3. When looking at memory contents, the debugger can tell us “you are looking at a 16bit waveform”.
4. This allows the debugger to export the samples in universally understood file formats like WAV or AIFF.

Detected Audio Data

1. We can create a list of all objects in memory
2. We can specify which types hold audio data and how to get metadata for them.
3. When looking at memory contents, the debugger can tell us "you are looking at a 16bit waveform".
4. This allows the debugger to export the samples in universally understood file formats like WAV or AIFF.
5. That requires knowing the sample rate, which may not be stored with the buffer and might need to be provided out-of-band.
6. Can we do more?

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.
3. The first thing to check is whether the problem exists in any of the inputs or only in the output.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.
3. The first thing to check is whether the problem exists in any of the inputs or only in the output.
4. We can extract the individual waveforms into files, open them in an external program and look at the spectrogram or play them.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.
3. The first thing to check is whether the problem exists in any of the inputs or only in the output.
4. We can extract the individual waveforms into files, open them in an external program and look at the spectrogram or play them.
5. We can configure the debugger to visualize a waveform and look at the spectrogram in the debugger.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.
3. The first thing to check is whether the problem exists in any of the inputs or only in the output.
4. We can extract the individual waveforms into files, open them in an external program and look at the spectrogram or play them.
5. We can configure the debugger to visualize a waveform and look at the spectrogram in the debugger.
6. We can write a Python script that detects high-frequency clicks and have the debugger run the script on the inputs.

High-Level Debugging

1. Let's say we are debugging a complicated function that mixes several input channels into one and resamples the output.
2. There are annoying high-frequency clicks in the output.
3. The first thing to check is whether the problem exists in any of the inputs or only in the output.
4. We can extract the individual waveforms into files, open them in an external program and look at the spectrogram or play them.
5. We can configure the debugger to visualize a waveform and look at the spectrogram in the debugger.
6. We can write a Python script that detects high-frequency clicks and have the debugger run the script on the inputs.
7. If you want to, you can ask the AI agent in your IDE to perform these tasks for you.

High-Level Debugging: Summary

1. The hardest part is modelling the world in useful abstractions. Your source code does that already.
2. Your program represents this model of the real world in bits and bytes.
3. A debugger allows you to see the bits and bytes of the program and to link machine code to source code.
4. I think a debugger should also translate bits and bytes back to the types specified in your source code.
5. We can then build new tooling that operates on these abstractions.
6. Different code bases can share the same tools if they use compatible abstractions.

End

Thank you for your attention.

Feel free to ask questions!

These slides are available at
`https://github.com/core-explorer/blog/blob/main/enumerate-audio.pdf`
The prototype is available at `https://github.com/core-explorer/core-explorer`