COMP2017 9017 Assignment 1

Due: 23:59 2 April 2025

This assignment is worth 10% of your final assessment

Contents

1 Assignment 1 - [SEGfault SOUNDboard] - 10%

We strongly recommend reading this entire document at least twice. You are encouraged to ask questions on Ed after you have first **searched**, and checked for updates of this document. If the question has not been asked before, make sure your question post is of type **"Question"** and is under **"Assignment" category** → **"P1"** . Please follow the staff directions for using the question template.

It is important that you continually back up your assignment files onto your own machine, flash drives, external hard drives and cloud storage providers (as private). You are encouraged to submit your assignment regularly while you are in the process of completing it.

Full reproduction steps (seed, description of what you tried) **MUST** be given if you are enquiring about a test failure or if you believe there is a bug in the marking script.

Academic Declaration

By submitting this assignment you declare the following: I declare that I have read and understood the University of Sydney Student Plagiarism: Academic Integrity Policy, Coursework Policy and Procedures, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Academic Integrity Policy, Coursework Policy and Procedures can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, Generative AI where approved, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I acknowledge that I have reviewed and understood the University of Sydney's guidelines on the responsible use of Generative AI [1] and will adhere to them in accordance with academic integrity policies.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the tutorial, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce

[1] https://www.sydney.edu.au/students/academic-integrity/artificial-intelligence.html

it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

2 Introduction

Audio is a digitised waveform representing a sound. The sound has a frequency, which can be encoded at a given sample rate, affecting the quality of the audio (bitrate). These properties are encoded as sequences of amplitude values over time in memory.

Editing audio involves various operations such as clipping, inserting, and moving. Clipping refers to selecting a portion of an audio file to keep or remove, inserting involves adding new portions at specific points, while moving would change a portion's relative position in time.

To support these operations, memory must be moved or copied and this can lead to inefficiencies. Instead, an audio editor's backend should use a shared backing store, where multiple operations reference the same underlying data.

3 Task

You will develop an audio editor backend that specialises in clipping and inserting data with a shared backing - where changes made will affect all portions that reference it. Users of this software will use the specified function prototypes to edit audio with simple operations. Your code should use data structures and algorithms to efficiently support editing as well as the ability to read and write to a buffer.

4 Structure

The audio data is sourced from a WAV file. The entire WAV file is read and stored into a buffer.

A track is a data structure that copies a continuous region of the buffer. A track can represent the entire audio or specific parts.

Any number of tracks can be created and the track can contain metadata that is useful to support the operations of this editor.

Each track is represented as an opaque data structure `struct sound_seg` that you must complete, according to the needs of *your* implementation. Each structure represents one audio track.

```c
// a track
struct sound_seg {
    // TODO
};
```

The audio editor exposes functions in section 5, which you will complete.

The functionalities of your program are divided into parts with varying levels of complexity. Each part has different requirements and is accompanied by specific assumptions. You should plan well for a particular level of achievement before coding. Writing helper functions are encouraged.

You are also required to answer the short questions described in section 6.

5 Functionality

5.1 Part 1: WAV file interaction, basic sound manipulation

Conversion between a sound file and a track is a two-step process involving an intermediate buffer.

Functions to interact between WAV files and a buffer

```
void wav_load(const char* fname, int16_t* dest);
```

The `wav_load()` function reads raw audio samples from the specified WAV file `fname` and copies them into the destination buffer `dest`. The WAV file's header is discarded during the loading process, leaving only the raw audio sample data in `dest`.

```
void wav_save(const char* fname, const int16_t* src, size_t len):
```

The `wav_save()` function creates or overwrite a WAV file, `fname`, using the audio samples provided in the source buffer `src`. The function constructs a valid WAV file, including the necessary header, and writes the audio samples to the file.
Note: this function does not free the memory pointed to by `src`.
You can find more about the WAV file format here.

REQ 0.1: An existing sound file can be loaded from/to a buffer.

Testing method: sanity

ASM 0.1: the song will always be PCM, 16 bits per sample, mono, 8000Hz sample rate.

ASM 0.2: the provided path for `wav_load()`, `wav_save()` will always be valid. IO operations are always successful. `dest` will be large enough.

All other functions **do not** require reading a WAV file, and can be operated with `int16_t` arrays.

Functions to interact between a buffer and a track

```
struct sound_seg* tr_init();
void tr_destroy(struct sound_seg* track);
```

`tr_init()` allocates and returns heap memory for a new empty track. This function may also initialise the structure to default values.

`tr_destroy()` releases all associated resources and deallocates the heap allocated pointer to `struct sound_seg`.

REQ 1.1: `tr_destroy()` should free all memory the track is responsible for.

Testing method: random

```
size_t tr_length(struct sound_seg* track);
```

`tr_length()` will return the current number of samples contained within this `track`.

```
void tr_read(struct sound_seg* track,
    int16_t* dest, size_t pos, size_t len)
```

`tr_read()` copies `len` audio samples from position `pos` in the `track` data structure to a buffer `dest`. `dest` is an externally allocated buffer with guaranteed size of at least `len`.

```
void tr_write(struct sound_seg* track,
    const int16_t* src, size_t pos, size_t len)
```

`tr_write()` function copies `len` audio samples from a buffer, `src`, to the specified position `pos`, within the data structure `track`. Any previous data stored in the track for the range of `pos` to `pos+len` is overwritten.

If the number of audio samples to be written to the track extend beyond the length of the track, the track's length is extended to accommodate the new data. Thus, a sequence of `wav_load()`, `tr_init()`, and `tr_write()` effectively transfers a WAV file to a track.

An ordering requirement when performing writes is to always write to lower indices before higher ones. This is only relevant for 5.3 onwards.

REQ 2.1: reads and writes make a copy of data from/to `buf`.

Testing method: sanity

REQ 2.2: `write` indices beyond the length of a track increases its length.

Testing method: random

You should make the functionality of `tr_init()`, `tr_destroy()`, `tr_length()`, `tr_read()`, `tr_write()` your first priority. As the marking script uses them to check the behaviour of other functions. Check section 7 for more details.

```
bool tr_delete_range(struct sound_seg* track, size_t pos, size_t len)
```

`tr_delete_range()` removes a portion from a track. The portion to remove begins at `pos` and spans `len` samples. After deletion, subsequent reads of this track return the samples just before `pos`, immediately followed by those at `pos + len`, effectively skipping the removed portion. On success, `delete_range` returns `true`. A failure case exists if `tr_insert()` (5.3) is implemented, and should return `false`.

Note: Samples removed by `tr_delete_range()` do not necessarily have to be freed from memory immediately, but should be freed when `tr_destroy()` is called.

REQ 3.1: reads and writes over deleted portion act as if adjacent parts are continuous.

Testing method: random
Prerequisites: REQ 2.2

Part 1 Checklist

1. Program is able to compile through makefile.
2. Using a main function, program is able to read a WAV file.
3. Using a main function, program is able to `wav_load()` and `wav_save()`.
4. Program is able to dynamically create empty tracks.
5. `length` and `read` is functional.
6. `write` is functional.
7. `delete_range` is functional.

5.2 Part 2: Identify advertisements

There are different kinds of sound represented as audio and as with all computer science problems, searching is essential. A search may identify a song of a bird in a natural setting, a musical tune, or even spoken words. Fortunately, there are algorithms such as Cross Correlation[2] that allow us to analyse two digital waveforms and compute their similarity. Effectively allowing you to determine if, and where, one sound appears in another.

In the modern world, audio media is often accompanied with an advertisement (ad). This is unwanted noise and we do not accept this. You will identify and remove these ads using Cross Correlation.

You are to create a function to search for the existence and locations of an ad within a target track.

```
char* tr_identify(const struct sound_seg* target,
                  const struct sound_seg* ad)
```

Returns a dynamically-allocated string in the format of `"<start>,<end>"` indicating the start and end indices of the `ad` occurrence in the target, inclusive. If there are no ads, return an empty string. If there are multiple ads, return a string consisting of all the index pairs, separated by a single newline character `\n`, such as `"<start0>,<end0>\n<start1>,<end1>\n<start2>,<end2>"`

REQ 4.1: `tr_identify()` is able to identify potentially-multiple, non-overlapping occurrences of `ad` in `target`.

Testing method: sanity
Prerequisites: REQ 2.2

Functionality is tested by directly overwriting portions of the `target` with copies of the `ad`, ensuring identical amplitudes. The `ads` will always have the same amplitude and there is no scaling needed.

Functionality is tested by copying multiple `ads` over `target`, with their amplitude values summed." Similarity is quantified by comparing correlation of the overwritten portion with the `ad`'s autocorrelation (cross correlation with the itself) at 0 relative time delay. As the reference, zero delay, this is 100% match. A portion is said to match if the `ad` is *at least* 95% of the reference [3].

ASM 4.1: The occurrences of ads in `target` will be non-overlapping and sufficiently clear.

[2] correlation with signals requires taking the complex conjugate, but as we are working with real signals, it can be ignored and individual samples simply multiplied.

[3] In the testcase, all correlation values larger than 95 are guaranteed to be ads. You do not have to consider the case where correlation is larger than 100.

The return method for `tr_identify()` function is poorly designed. You may be asked to address this issue with an explanation. See section 6 for more details.

Part 2 Checklist

1. Able to compute autocorrelation, the reference.
2. Able to compute cross-correlation and return string value(s).

5.3 Part 3: Complex insertions

The true value of the editor backend comes from mixing and clipping audio.

```
void tr_insert(struct sound_seg* src_track,
               struct soung_seg* dest_track,
               size_t destpos, size_t srcpos, size_t len);
```

`tr_insert()` performs a *logical* insertion of a portion from a source track into a destination track. The portion to be inserted are `len` samples beginning at position `srcpos` in `src_track`. The insertion point is at position `destpos` in `dest_track`.

After insertion, `dest_track`'s data before `destpos` remains unchanged, followed by the inserted portion, and then the remaining original data from `dest_track`.

Note: This function is conceptually the inverse of `delete_range()`.

This consequence of a `tr_insert()` operation results in a parent-to-child relationship. The parent (src) and the child (dest) portions should have shared backing store and the data need only be stored once, **saving memory**. Further `insert()` operations performed on the parent or child similarly extend this shared backing, such that `tr_write()` to one sample in a portion of one track could result in changes across many other tracks. As `tr_delete_range` and future `tr_inserts` do not change track data but track structure, their changes are not propagated.

Note: for cases of self-insertions. The portion is determined at the time `tr_insert()` is called, before the portion is inserted. Thus, inserting a portion into oneself is well-defined.

REQ 5.1: `tr_insert()` inserts a reference copy of `src`'s portion into `dest`

Testing method: random*. Due to complexity of this function, extra tiered restrictions have been laid out - you may find that they significantly decrease programming complexity:
5.1.1: Every sample in the parent to be inserted, and samples adjacent `destpos`, shall not already be a parent or child themselves.
5.1.2: Every sample in the parent to be inserted shall not already be a parent or child themselves.
5.1.3: Samples adjacent `destpos` shall not already be a parent or child.
5.1.4: Samples adjacent `destpos` shall not already be a parent.
5.1.5: Every sample in the parent to be inserted shall not already be a child.
5.1.6: No restrictions.
Prerequisities: all other requirements

Because the function `tr_insert()` operates on the same track, other functions must have stricter requirements for the function to operate correctly.

For functions `tr_read()`/`tr_write()`:

REQ 2.3: changes (write) to a child portion must be reflected in the parent, and vice versa

Testing method: random

For functions `tr_delete_range()`/`tr_destroy()`:

REQ 3.2: A parent portion may not be deleted if it has children. Attempts to do so return `false`. `tr_destroy()` nonetheless removes the portion.

Testing method: random

ASM 0.3: `tr_destroy()` will only be called at the end of the program, on all tracks to free memory.

You should use a linked data structure to implement `tr_insert`.

Part 3 Checklist

1. Implement the trivialised version of `tr_insert()` by copying sample data (wasteful data duplication).
2. Understand and model the behaviour of `tr_insert()`.
3. Implement `tr_insert()` at **5.1.1** level.
4. Ensure requirements for other functions hold.
5. Implement `tr_insert()` at **5.1.6** level.

5.4 [COMP9017 ONLY] Part 4: Cleaning Up

Too many `tr_insert()` operations can lead to confusing parent-child relationships. The following function aims to alleviate this issue.

```
void tr_resolve(struct sound_seg** tracks, size_t tracks_len);
```

`tr_resolve()` conditionally breaks parent-child relationships for specified tracks. Given an array of track references `tracks`, if a portion *Pi* is a direct parent to another, *Pj*, and both portions can be found in `tracks`, this will break their relationship, such that:

- *Pj* is no longer a child.
- *Pi* is no longer a parent if it does not have other children.

In the trivial case, if both *Pi* and *Pj* exist in track *T* and `tr_resolve` was called on *T*, the track will effectively be flattened and the previously shared memory of those portions becomes duplicated data.

Consider tracks `A`, `B`, `C`, `D`, `E` with a shared portion between them and the corresponding parent->child relationships as `A->B`, `B->C`, `C->D`, `A->E`. If `tr_resolve` was called on `{B, C}`, then after calling `tr_resolve()`:

- `B->C` no longer exists.
- `A->B` still exists, as `A` was not provided. By similar logic, `C->D` also exists.

- `A->E` still exists, as neither `A` nor `E` were provided.
- The portion in `B` can now be `delete_range`'d, as it is no longer a parent.
- `A` is a parent maintaining the portion (as before)
- `C` becomes a parent maintaining the portion (duplicated as a result of breaking from `B`)

`tr_resolve()` has now effectively split the shared backing store into two. The portions in `A`, `B`, `E` in one, and `C`, `D` in another.

If `tr_resolve()` was called on `{A, C}` or `{A, E}`, although they share the same memory backing, nothing will happen as they do not have a direct parent-child relationship.

REQ 6.1: `tr_resolve()` removes every direct parent-child relationship if the list provided contains both parent and child.

> Testing method: random.
> Test case is private. Please write your own to verify.
> Prerequisites: REQ 5.1

5.5 Performance

Memory usage and leaks are tracked in your program by dynamically replacing symbols `malloc`, `calloc`, `realloc` and `free`.[4] You should only use the above standard dynamic memory allocation functions.

Random testcases for `tr_insert()` enforce a max dynamic memory usage.

5.6 Global assumptions

To simplify logic, you can ignore index bounds checking.

ASM 7.1: indices covered by `tr_read()`, `tr_delete_range()`, and `srcpos` and `len` in `tr_insert()` are always in range.

ASM 7.2: The starting position for `tr_write()` and `destpos` for `tr_insert()` ranges from 0 to the target track length, inclusive.

6 Short answer questions

As part of the in-tutorial code review in week 8, you are required to analyse your code and prepare for **two** of the below questions. You must supplement your answer with references to your code. The examiner will also ask follow-up questions based on your response. COMP9017 students must answer Q4.

Q1: How may you redesign the function prototype for `identify`, such that it more robustly returns the list of ad starts and ends?

[4] Note that some functions, like `printf`, also use dynamic memory. Do not call them in your submission.

Q2: Referring to REQ 1.1, how did you identify which track is responsible for which memory, and how did you ensure that all memory is freed? If you were not successful in ensuring, how did you plan to?

Q3: **[COMP2017 ONLY]** Explain the time complexity of your `tr_insert()` and `tr_read()` by referring to the relevant parts of your source code.

Q4: Demonstrate how you constructed test cases and the testing methods used to confirm your program functions correctly. If you answer this question, the testcases must be in your final submission in a folder named `tests`, and all tests should be run by the file `tests/run_all_tests.sh`.

7 Marking

7.1 Compilation requirements

Using the `make` program, your submission should compile into an object file, which the user/marker will utilise.

Your submission must produce an object file named `sound_seg.o` using the command `make sound_seg.o`. The marking script will compile this into a shared library to be used. Thus, the flag `-fPIC` must be added.

You are free to (and encouraged to) add extra build rules and functions for your local testing, such as a `main` function or debug flags. ASAN is encouraged during local testing, **and will be automatically added to your final submission**. [5]

When marking your code will be compiled and run entirely on the Ed workspace. The marker will run the aforementioned `make` commands to compile your program and run the executable. If it does not compile on the environment, then your code will receive **no marks for your attempt**. When submitting your work ensure that any binary files generated are not pushed onto the repository.

7.2 Test structure

After your object file is compiled into a shared library, python scripts (`ctypes`) are used to interact with the functions described in spec. In most cases, the script is responsible for:

- Creating temporary data,
- Orchestrating calling of functions,
- Comparing returned data with expected values.

This is used for both sanity and random tests. Thus, you can think of the test inputs and outputs as not given from a separate program (and waits for you own program to respond and exit), but rather driven in the same program, and the outputs are validated before your program ends.

[5]The marking script will attempt to **add** ASAN and PIC during compilation by appending the flags `-fno-sanitize=all -fPIC -Wvla -Werror -fsanitize=address -g`. If this is not successful, marking will silently fail.

7.3 Seeded testcases

All `*_random` testcases have the following structure:

1. random amount of tracks are created.
2. a random array is written to each track using `tr_write()`.
3. a random operation between `tr_write()`, `tr_delete_range()`, and `tr_insert()` is chosen if allowed.
4. `tr_length()` and `tr_read()` is done on random tracks and verified against expected values.
5. repeat random operation and verification for some cycles.
6. all tracks are properly managed where `tr_destroy()` is called and implemented correctly. Memory leak check.
7. return value is checked (non-zero indicates failure). [6]

If a failure is reached, the marking script attempts to return the input set that caused the failure, which you can use locally to debug. Additionally, you are also able to manipulate random testcases for your own testing - details have been provided in the EdStem lesson.

For each random testcase in a submission, the seed used is included in the feedback section and can be used to deterministically regenerate inputs. During the marking phase, a predetermined set (15+) of seeds will be used and the percentage passed will become your final mark for a specific test. The assignment EdStem lesson provides more details for configuring random testcases.

From rudimentary analysis, passing `insert_no_overlap_*_random` for a **single** seed implies you will also pass 95% of other seeds, and passing other random tests for a single seed implies 99+%. If you only submit **once** and all 7 random testcases pass, you would expect a HD mark with very low variance. Submitting more than once, and thus testing using multiple seeds, greatly increases this confidence level; but even if you only submit once, the confidence of passing the reserved seeds far exceed the confidence of passing a private testcase if only a static testcase is used.

All final test inputs will be posted after 17 April.

7.4 Marking criteria

The assignment is worth 10% of your final grade. This is marked out of 20, and breaks down as follows. For marks awarded per testcase, please refer to Edstem.

Marks	Item	Notes
$3/20$	Code Style	Manual marking
$5/20$	5.1 Correctness	Automatic tests
$4/20$	5.2 Correctness	Automatic tests
$8/20$	5.3 Correctness	Automatic tests

For style, refer to the style guide. You will also be marked based on the modularity and organisation of your code. For full marks, code should be organised in multiple source files, and use modular,

[6]**Thus, please don't return a nonzero value upon program exit.**

task-specific functions. Organised data structures are essential here. Style marking is only applied for reasonable attempts (5.1 Correctness).

[COMP9017 ONLY] 9017 students will have their above marks scaled by 0.9. 5.4 Correctness counts for 2/20.

7.5 Restrictions

To successfully complete this assignment you *must* (submissions breaking these restrictions will receive a deduction of up to 6 marks *per breach*):

- The code must entirely be written in the C programming language.
- Must use dynamic memory for tracks.
- Free all dynamic memory that is used.
- *NOT* use any external libraries other than those in `libc`.
- *NOT* use VLAs.
- *NOT* have unclean repositories. This means no object, executable, or temporary files **for any commit** in the repository, just your final submission.
- Only include header files that contain declarations of functions and extern variables. Do not define functions within header files.
- Must use meaningful commits and meaningful comments on commits. [7]
- Other restricted functions may come at a later date.
- Any and all comments must be written only in the English language.
- *NOT* manually use return code 42, reserved by ASAN.

The red flag items below will result in an immediate zero. Negative marks can be assigned if you do not follow the spec or if your code is unnecessarily or deliberately obfuscated:

- Any attempts to deceive or disrupt the marking system.
- Use any of the below functions. You shouldn't need to use these functions at all in your program, and you are doing something terribly wrong if you are.
  - `_init, atexit(2), _exit(2), _Exit(3)`
  - `dlopen(3), dlsym(3), dlclose(3)`
  - `fork(2), vfork(2), execve(2), exec*(3), clone(2)`
  - `kill(2), tkill(2), tgkill(2)`
  - `getpid(2), getppid(2), ptrace(2), getpgrp(2), setpgrp(2)`

8 Submission Checklist

- Submission have a valid makefile with the rule `sound_seg.o` and compiles.
- Reviewed all restrictions (not all are automatically checked)
- Program is organised into multiple source and header files (for larger programs).
- Not include any object file, binary, or junk data in your git repo.
- If you have used AI, `references.zip` formatted according to EdStem slides submitted with source code.

[7] "forcing the seed of a testcase" does not count as valid commit. Must cite reason and identified failure.

Glossary

assumption shortened: ASM. A property that is externally guaranteed to be true when your program is run. When testing, situations which violate this property will not happen. Thus, handling behaviour that falls outside of an assumption (e.g. out of bounds `read`) will not give you marks. 4

child A portion that has been `insert`ed from another part of a track. The portion is the child to the portion that it was copied from. A sample may only belong to one parent. `writes` to the child must be reflected in the parent. 8–10, 14

parent A portion that has been `insert`ed into another part of a track. The portion is the parent to only portions that exist due to that `insert`. A portion may be a parent to multiple children. `writes` to the parent must be reflected in the childen. After inserting, it is possible for a parent portion to be not contiguous. 8–10

portion refers to a part of a track. Contains zero or more samples. Portions are defined logically rather than their indices in a track. Indices of portion samples may change if a `delete_range` or `insert` modifies the length of the track.. 6–8, 14

random Property-based testing that test for the specified requirement, with inputs restricted by assumptions. In this assignment, the python library hypothesis is used. 5, 8, 11, 12

requirement shortened: REQ. A property that your program is expected to hold when run. Marks are given depending on how well your program holds them. Most requirements in parts 2 and 3 have prerequisites, properties that need to hold before the the current requirement is considered. 4

sample Audio is a digitised waveform representing a sound. The sound has a frequency, which can be encoded at a given sample rate. A sample is simply a numberic value representing the strength of sound at a particular time. In the context of this assignment, the data type for a sample is **`int16_t`**.. 5, 7, 14

sanity A directed testcase targeting a specific functionality. For example, a sanity test for REQ 2.1 may be to create a track, `write` into it, modify the original buffer, then verifying if the buffer and the track contents are different. Randomness may still be involved. 5, 11

shared backing store A shared backing store is a memory management technique where multiple references to the same underlying data are used instead of copying or moving memory. . 4, 8, 10, 15

track A **`struct sound_seg`** object. It represents the user's view of the API as users mix the different objects together. 4–7, 10–14

9 Appendix

9.1 Worked function example

Figure 1: This example uses two tracks. They are created and filled using a sequence of `tr_init`, and `tr_write` of data.

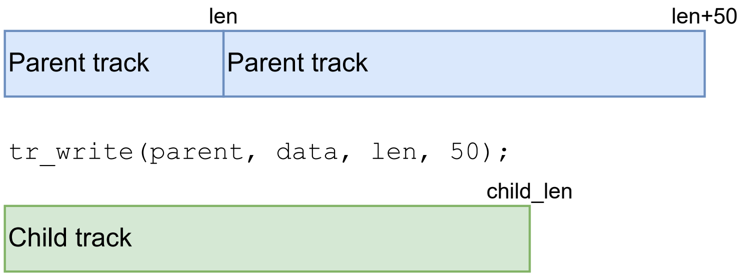

Figure 2: Either track can be extended via a call to `tr_write`. By calling write on the end of the parent, new data is effectively concatenated.

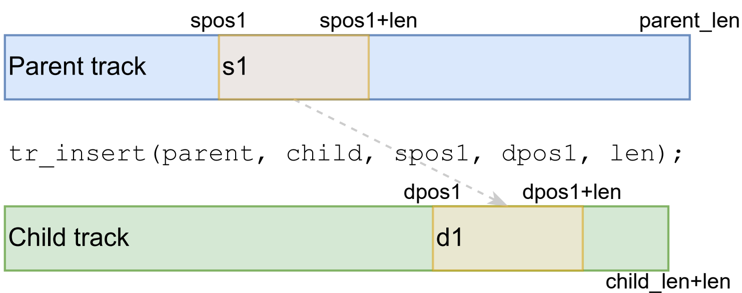

Figure 3: The initial insert extracts a portion `s1` from the parent, and places the portion into the child, also extending it. Due to shared backing store, there is a logical relationship between `s1` and `d1`.

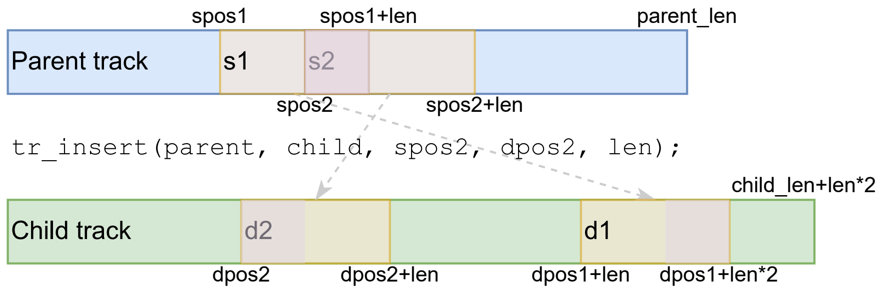

Figure 4: A second, overlapping insert occurs, placing `d2` before `d1`. Note that **1)** while the child is extended and indices for `d1` changed, the logical relashionship remains. **2)** the overlapping samples of `s1` and `s2` means that parts `d1` and `d2` (highlighted in purple), even though unrelated, also share samples.

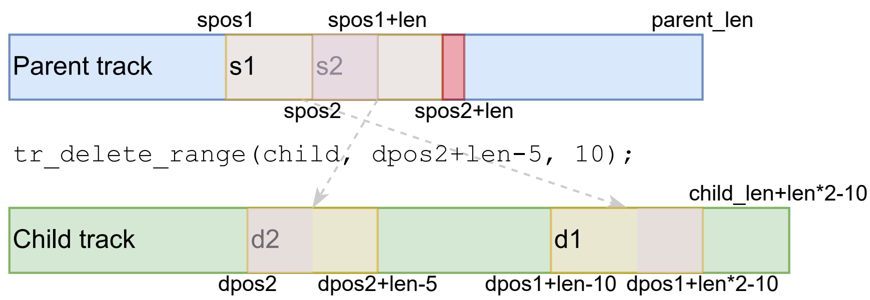

Figure 5: `tr_delete_range` will fail if any of the specified samples is a parent (in this case, `s1` and `s2`. Child samples such as a part of `d2` can still be deleted (the command deletes the last 5 samples of `d2`, and 5 samples after the end, for 10 total). Because `d2` no longer contains the last 5 samples, The last 5 samples of `s2` (in red) also stops being a parent; there is no immedate change, but those samples can now be deleted. Again noticed how the indices for `d1` were shifted without impacting the parent-child relationship.

10 Version history

We aim to resolve all spec updates within the first 3-5 days.

22/03/2025-23:07

- Clarified matching criteria for `tr_identify`.
- Removed **`const`** qualifier from `tr_insert`.
- **Changed ASAN requirement from strictly forbid to strictly allow.**

19/03/2025-11:12

- Clarify that most functions only interact with **`int16_t`** buffers, not WAV files, multiple times in the spec
- clarified definition of sample, in the case of this assignment analogus to **`int16_t`**.
- If you plan to answer Q4 short answer, you must upload a folder called `tests` with your tests in them.
- Added some banned function restrictions, which already exist in the testcase.
- Added prerequisite to `tr_resolve`
- Reworded `tr_identify` from "ads overwriting target" to "ads inserted on top of target". Such that the ads in the target aren't exactly the same.

14/03/2025-10:35

- Created version history.
- Added detailed description of marking process with python.
- Reword `dest` in REQ 5.1 to `destpos`.
- correct return value of `tr_destroy` from **`bool`** to **`void`**.
- Define what an unclean repo is.
- Clarfied that code must be written in C, and compile in EdStem.
- Improve wording of `tr_resolve` from "previously shared memory becomes duplicated data" to "previously shared memory of those portions becomes duplicated data", to clarify only specific portions are flattened.
- Created submission checklist.
- Added suggestions of extra makefile rules for students' own testing.
- Added linked to EdStem slides about manipulating EdStem testcases, and submitting AI references.
- Added `-Wvla -Werror` as implicit compilation flags.