CS 240: Programming in C
Fall 2020
Project

Prof. Turkstra

Checkpoint 1 Due: November 8, 2020 9:00 PM
Checkpoint 2 Due: November 22, 2020 9:00PM
Checkpoint 3 Due: December 6, 2020 9:00 PM

1 Goals

The purpose of this project is to serve as a comprehensive analysis of the skills you should have acquired thus far, entailing the use of file IO, dynamic memory allocation, and dynamic structures such as trees and lists.

Note on `malloc()`

It is important to check the return value of `malloc()`. Make sure that you always `assert()` that the address returned by `malloc()` is not NULL.

Now that we are dealing with dynamic memory allocation, we have a special malloc debug library that you can use if you are having trouble. Simply replace

```
#include <malloc.h>
```

with

```
#include <malloc_debug.h>
```

and recompile.

You should not include `stdlib.h`, but rather `malloc.h`.

2 Getting Started

1. Log in to one of the Purdue CS Linux systems. These include data.cs.purdue.edu and borgNN.cs.purdue.edu where NN=01,02, etc
2. Inside the cs240 directory, setup a proj1 directory by running the following commands...

```
$ cd cs240
$ git clone ~cs240/repos/$USER/proj1
$ cd proj1
```

3. This creates your proj1 directory, and also copies what is called a **Makefile** into it. This Makefile has a number of targets defined to make your life easier...

   When you are ready to try your work, you can run:

   ```
   $ make proj1_main_p1
   $ make proj1_main_p2
   ```

   to build main for checkpoint 1 and checkpoint (2 and 3), respecitively, or run:

   ```
   $ make
   ```

   to build both main and test.

   For this project, the test module will not be released at the beginning and we will announce it when the test module is ready.

3 The Big Idea

This project will be taking the place of regular tests, and will accordingly be both broader in scope and less guided than the standard programming assignments. However, upon the conclusion of the project, you will be left with something you can hopefully point to and say "that's neat."

The main idea behind this project is to recreate a rudimentary music library with associated analysis and adjustment tools.

To discourage people from making any bad time management decisions, this project will be broken down into multiple parts with sporadic graded checkpoints. Each section will correspond to different .h/.c file pairs, and later sections will build on your work in prior ones. Thus, even if you don't get some parts of your code finished by the checkpoint due date, you'll still need to get them working for later parts to run correctly. To avoid overloading you with information early on, the following section of the handout will discuss each section individually in the order that the assignment should be completed. Additionally, given the increased volume of data structures required, specific definitions will be omitted from this document. Thus, it is highly recommended that you skim through the associated header file prior to reading the handout's instructions for that section, and that you leave the header file open for easy reference.

4 The Assignment

4.1 MIDI Parsing (Checkpoint 1)

4.1.1 MIDI Files

This section of the assignment will take place in "parser.*", and "event_tables.*" (though you will only need to modify "parser.c"). For this part of the assignment, you will implement logic to parse MIDI files into `song_data_t` structs. For an in-depth look at the structure of MIDI files, we recommend consulting this website, but you are welcome to use other resources as well to help you understand the format, provided of course they supply a correct description of the file format.

As a brief summary of the format, MIDI files are music files for storing electronic music specifications (in a binary format). They contain various chunks, the first of which is a header which will have various metadata including timing information that should be parsed directly into the `song_data_t` structure. Following the header will be one or more track chunks, each of which are comprised of variable numbers of different types of events. The `track_node_t` and `event_node_t` structures will act as the nodes of associated linked lists that house track and event (`track_t` and `event_t`) information respectively.

Events come in three flavors: sysex events (`sys_event_t`), meta events (`meta_event_t`), and MIDI events (`midi_event_t`). The first is for passing system exclusive information, and will not be of particular interest for our purposes. Meta events are named events that contain information more useful information like time/key signatures, but given that our program will not actually play the songs itself, we will not be focussing on these events either. MIDI events are named events that contain more specific information about what notes are being played at what times, and in what fashion. These will be the events of greatest interest to us in later parts of the assignment, and accordingly have the most complex representations. These events are distinguished from one another by their leading byte (denoted `type` in `event_t`), and contain subsequent to that byte `data_len` additional bytes to be stored in their `data` array.

To aid you in parsing events, we supply `META_TABLE` and `MIDI_TABLE` (see "event_tables.*"), which are constructed in `build_event_tables()` prior to when the `main()` function runs. These tables map from the sub-type of events to a default event of the relevant type (this will make more sense after consulting the prior link to MIDI formats). For events with a fixed size, that size is recorded in the `data_len` attribute of the default structure. For events with a variable size, the `data_len` attribute is marked as 0 in the default structure. The use of these tables is encouraged, though not mandatory. If you wish to use your own parsing schema, feel free, so long as you return appropriate data.

In addition, to help you debug your functions as you're working, we provide access to `write_song_data` (as described in "song_writer.h"): a function that writes a `song_data_t` struct in MIDI format back to a given file. You may use this to perform loopback tests, in which you parse a MIDI file, then write your parsed version back to a different file and compare the results. If your code is implemented correctly, there should be no difference between the file you parse, and the file that is written back.

4.1.2 Functions to Implement

We **strongly** recommend looking over the description of each function prior to attempting to write any of them. Many of the functions will make use of one another, so spend some time thinking about what your solution will look like before trying to code it so that you don't waste time trying to implement the same thing in multiple places.

`song_data_t *parse_file(const char *);`

This function takes in the path of a MIDI file and returns the parsed representation of that song (be sure to dynamically allocate all structures as necessary so that they will persist after this function returns).

Be sure to deep copy the path, and remember that MIDI is a binary file format. **You should NOT attempt to implement all of the parsing logic in this function**. Following

functions will implement sections of the parsing logic that should be called within this function.

Assert that the input pointer is not null, that the file can be read, and that, after your parsing logic has finished, there is no additional data in the file.

`void parse_header(FILE *, song_data_t *);`

This function should read a MIDI header chunk from the given file pointer and update the given `song_data_t` pointer with the format, number of tracks, and division information.

Assert that the chunk type is correct (i.e. that the first 4 bytes read from the file pointer are "MThd"), that the format, number of tracks, and division information occupy 6 bytes in total, and that the format is valid (see Section 2.1 of the aforementioned website).

`void parse_track(FILE *, song_data_t *);`

This function should read a MIDI track chunk from the given file pointer and update the given `song_data_t` pointer's `track_list` with the `track_t` extracted from the file.

Assert that the chunk type is correct.

`event_t *parse_event(FILE *);`

This function should read and return a pointer to an `event_t` struct from the given MIDI file pointer. For meta events, the `type` attribute of the event should be `META_EVENT`; for sysex events, the `type` should be either `SYS_EVENT_1` or `SYS_EVENT_2`; for midi events, the type should be set to the status of the event.

Based on the type of the event, its union member should be set accordingly.

`sys_event_t parse_sys_event(FILE *);`

This function should read and return a `sys_event_t` struct from the given MIDI file pointer. Be sure to allocate enough space in the `data` pointer to house all of the event data (event data starts after the length specification). `data_len` should encode the size of the `data` array.

`meta_event_t parse_meta_event(FILE *);`

This function should read and return a `meta_event_t` struct from the given MIDI file pointer. Be sure to allocate an appropriate amount of space in the `data` pointer. However, given that event names should not change, you should leave the `name` pointer as the default value so as not to waste space.

For meta event types that have a known, fixed length, you should assert that the length specified in the file is equal to the known length. Additionally, you should assert that the event found is of a valid meta event type. Invalid events have no entry in `META_TABLE`. Hint: What is the value of the `name` attribute for events not present in the `META_TABLE`?

`midi_event_t parse_midi_event(FILE *, uint8_t);`

This function should read and return a `midi_event_t` struct from the given MIDI file pointer. Be sure to allocate an appropriate amount of space in the `data` pointer. However, given that event names should not change, you should leave the `name` pointer as the default value so as not to waste space.

Read Section 2.3 of the aforementioned website carefully, as sometimes the status byte is encoded implicitly. To determine whether the status was encoded implicitly or not, think about

what can be used to reliably differentiate between status and data bytes. Hint: it's one of the bits. You may wish to use a global or static variable to help you with this function.

You should assert that the event found is of a valid type. Invalid events have no entry in `MIDI_TABLE`.

`uint32_t parse_var_len(FILE *);`

This function should read a variable length integer from the given MIDI file pointer and return it as a fixed-size `uint32_t`.

`uint16_t end_swap_16(uint8_t [2]);`

This function takes in a buffer of two `uint8_t`s representing a `uint16_t`, and should return a `uint16_t` with opposite endianness.

This function will be useful for parsing certain numbers from the file, as the endianness used by MIDI does not always align with that of modern systems.

`uint32_t end_swap_32(uint8_t [4]);`

This function takes in a buffer of four `uint8_t`s representing a `uint32_t`, and should return a `uint32_t` with opposite endianness.

This function will be useful for parsing certain numbers from the file, as the endianness used by MIDI does not always align with that of modern systems.

`uint8_t event_type(event_t *);`

This function takes in a pointer to an event, and should return either `SYS_EVENT_T`, `META_EVENT_T`, or `MIDI_EVENT_T`, based on the `type` of the event.

`void free_song(song_data_t *);`

This function should free all memory associated with the given song (including the track list).

`void free_track_node(track_node_t *);`

This function should free all memory associated with the given track node (including the track and its associated events).

`void free_event_node(event_node_t *);`

This function should free all memory associated with the given event node (including the event and any allocated data arrays).

4.2 MIDI Library (Checkpoint 1)

4.2.1 Library Format

This section of the assignment will take place in "library.*", and will involve creating a binary tree to act as a library of for MIDI songs. Each `tree_node_t` will house a song, and will use the name of the song as its key. Key comparisons should be performed using `strcmp`. Additionally, it should be noted that a song's name is **not** the same as it's path. In particular, a song's name

should not include any directory information. Thus, the song name for "./songs/album/xyz/my_song.mid" would be "my_song.mid". Given that the song name is a substring of the path, however, no additional space should be allocated for the name. Rather, `song_name` should simply point to the starting location of the song name in the `path` string.

4.2.2 Functions to Implement

`tree_node_t **find_parent_pointer(tree_node_t **, char *);`

This function should locate the node in the given tree having the given `song_name`. If the desired song cannot be found in the tree, return `NULL`. Otherwise, return the parent node's pointer to the desired song. For example, if the desired node is the left child of the root node, the value returned should be the address of the the root node's `left_child` pointer.

If the root node itself is the desired pointer, you should return the input pointer.

`int tree_insert(tree_node_t **, tree_node_t *);`

The first argument to this function is a double pointer to the root of the tree. The second argument is a node to be inserted into the tree. If the node is already present in the tree, return `DUPLICATE_SONG`. Otherwise, insert the node at the appropriate location and return `INSERT_SUCCESS`.

`int remove_song_from_tree(tree_node_t **, char *);`

This function should search the given tree for a song with the given name. If no such song is found, return `SONG_NOT_FOUND`. Otherwise, remove the `tree_node_t` associated with that song from the tree and free all memory associated with the node, including the song. Then, return `DELETE_SUCCESS`.

If the removed node had any children, remember to re-insert them into the tree afterwards.

`void traverse_pre_order(tree_node_t *, void *, traversal_func_t);`

This function accepts a tree pointer, a piece of arbitrary data, and a function pointer. This function should perform a pre-order traversal of the tree, where the operation applied to each visited node is a call to the function pointer with the node and arbitrary data passed in.

`void traverse_in_order(tree_node_t *, void *, traversal_func_t);`

This function is exactly the same as the prior, but should visit each node in an in-order fashion.

`void traverse_post_order(tree_node_t *, void *, traversal_func_t);`

This function is exactly the same as the prior, but should visit each node in a post-order fashion.

`void free_node(tree_node_t *);`

This functions should free all data associated with the given `tree_node_t`, including the data associated with its song.

`void print_node(tree_node_t *, FILE *);`

This function should print the song name associated with the given tree node to the specified file pointer, followed by a newline characater.

`void free_library(tree_node_t *);`

This function should free all data associated with the input tree.

`void write_song_list(FILE *fp, tree_node_t *);`

This function should print the names of all songs in the given tree to the file, separated by newline characters. These songs should be printed in sorted order, where the sorting criteria is based on `strcmp`. Hint: this can be done in one line using prior functions...

A point of interest is that `stdout` is, essentially, a variable associated with the open file pointer of the terminal you're running your code in. This may be of use to you as you test your code.

`void make_library(const char *);`

This function takes in the name of a directory, and should search through the directory structure, adding adding every MIDI file found in the structure (any file with the extension ".mid") to the tree held by `g_song_library`. You should assert that the directory structure contains no duplicate songs.

Hint: You may find the `ftw` function to be of use here.

4.3 Song Augmentation (Checkpoint 2)

4.3.1 The General Idea

This section of the assignment will take place in "alterations.*", and will focus on inspecting and transforming the songs created in the prior sections. Note that you may find some parts of this section less guided than before, in that we have not provided implementation guidelines for a number of useful helper functions. That said, you are still strongly encouraged to create helper functions of your own for some of the more complex functions in this section.

In order to properly appreciate the transformations you'll be inflicting upon the songs, we recommend familiarizing yourselves with the `scp` command, so that you may transfer the modified songs to your local machine and listen to them in your preferred music playing software.

4.3.2 Functions to Implement

`int apply_to_events(song_data_t *, event_func_t, void *);`

This function takes in a song, a function pointer, and a piece of arbitrary data, and applies that function with that piece of data to every event in the given song.

This function should return the sum of all the function return values over events in the song.

`int change_event_octave(event_t *, int *);`

This function takes in an event pointer and a pointer to a number of octaves. If the event is a note event (Note On, Note Off, Polyphonic Key Pressure), it changes the octave of that note by the given number of octaves, unless doing so would make the note value less than 0 or greater than 127.

note value: check appex1.1

This function returns 1 if the event was modified, and 0 otherwise.

`int change_event_time(event_t *, float *);`

This function takes in an event pointer and a pointer to a float multiplier. This function should scale the delta-time of the event by the input multiplier.

This function should return the difference in bytes associated with the variable length quantity representation of the event's new delta-time relative to its old delta-time. For example, if the old delta-time was `0x7F`, and the new delta-time is `0x80`, this function should return 1.

`int change_event_instrument(event_t *, remapping_t);`

This function takes in an event pointer and a table mapping from current instruments to the desired new instruments. If the event is a program change event (changing which instrument is in use), it modifies the instrument to be the value mapped to in the remapping table.

This function returns 1 if the event was a program change event, and 0 otherwise.

`int change_event_note(event_t *, remapping_t);`

This function takes in an event pointer and a table mapping from current notes to the desired new notes. If the event contains a note, it changes the note to the value mapped to in the remapping table.

This function returns 1 if the event contained a note, and 0 otherwise.

`int change_octave(song_data_t *, int);`

This function takes in a song and an integer number of octaves. Each note in the song should have its octave shifted by the given number of octaves.

This function should return the number of events that were modified.

`int warp_time(song_data_t *, float);`

This function takes in a song and a float multiplier. It should modify the song so that the overall length of the song changes by the multiplier. For example, if passed 0.5, the song should play twice as quickly as before. length * coef? for track or event?

Additionally, the song should be left in a correct state with regard to stated track lengths. That is, if the modified tracks will require more or less space when written to a midi file, their `length` parameter should be updated accordingly.

This function should return the difference in the number of bytes between the song's new representation when expressed as a midi file, and its original representation when expressed as a midi file.

`int remap_instruments(song_data_t *, remapping_t);`

This function takes in a song and a table mapping from current instruments to desired new instruments. It should modify the song so that all instruments used are remapped according to the table. For convenience, we have provided some simple tables to get you started (`I_BRASS_BAND` and `I_HELICOPTER`), but feel free to add your own.

This function should return the number of events that were modified.

`int remap_notes(song_data_t *, remapping_t);`

This function takes in a song and a table mapping from current notes to new notes. It should modify the song so that all notes are remapped according to the table. For convenience, we have provided some simple tables to get you started (`N_LOWER`), but feel free to add your own.

This function should return the number of events that were modified.

```
void add_round(song_data_t *, int, int, unsigned int, uint8_t);
```

This function takes in a song, a track index (int), an octave differential (int), a time delay (unsigned int), and an instrument (represented as a uint8_t per appendix 1.4 of the website), and uses them to turn the song into a round of sorts. To accomplish this, duplicate the track at the corresponding index in the song's track list (assume the list is 0 indexed), change its octave by the specified quantity, **delay** it by the specified value, and convert all of its instrumentation to use the specified instrument, then add it to the end of the song's track list.

nt time?

Make sure to update the song's metadata appropriately (number of tracks **and** format), as well as the created track's length. Hint: what operation might change the length of the track?

Additionally, to allow both the added track and the prior tracks to be heard, you should change the channel of all MIDI channel events (see appendix 1.1) in the new track to be the smallest channel value **not** found in the song.

Assert that the track index may be found within the song, that the song is not of format 2 (in which case adding a round doesn't really make sense), and that the song has not already used all of its channel values.

4.4 GUI (Checkpoint 3)

4.4.1 Making an Application

High level idea

This section of the assignment will take place in "ui.*". You are going to implement a graphical user interface (GUI) for the song library you created in Section 4.2 so that users can more easily take advantage of the functions in Section 4.3 to play with their favorite songs. The GUI will allow the user to:

- Add songs to the library
- Browse information about songs in the library
- Edit songs and visualize the changes in real time
- Save modified songs for later enjoyment

For the sake of simplicity, when a user selects a song, the associated node from `g_song_library` should be saved in `g_current_node`. When editing the song, all modifications should be done on a deep copy of the original song, which should be stored in `g_modified_song`. Only after the user confirms the modifications should the modified song been saved to disk.

GTK Introduction

GTK (GIMP Toolkit) is a library for creating graphical user interfaces. It provides an application programming interface (API) in many languages including C, C++, Guile, Perl, Python, TOM, Ada95, Objective C, Free Pascal, Eiffel, Java and C#. GTK is also cross-platform, being supported by many mainstream operating systems.

GTK is a widget toolkit, meaning that each user interface created by GTK is comprised of widgets. The window widget is the main container. The user interface is then built by adding buttons, drop-down menus, input fields, and other widgets to the window.

In addition, GTK is event-driven. This means that the toolkit listens for events, such as button clicks, and passes the events to your application through callback functions registered to various widgets.

GUI programming will be covered in lecture, but with any large framework, it's impossible to go over every feature you might find yourself in need of. Accordingly, we recommend checking out the following webpages for more information about GTK:

- `https://developer.gnome.org/gtk3/stable/` This is the documentation for GTK, which provides specifications for all the functions, as well as good example code.
- `https://developer.gnome.org/gtk3/stable/gtk-getting-started.html` This page contains some starter code along with detailed explanations. It should serve as a good starting point for you to familiarize yourself with GTK.
- `https://developer.gnome.org/gtk3/stable/ch03.html` This page showcases the most frequently used widgets in GTK, which you may find useful in your development efforts.

Example UI

Figure 1 shows an example implementation of what your application might look like. The panel on the left deals with the song library (using `g_song_library`), whereas the panel on the right is for examining and modifying particular songs (using `g_current_node` and `g_modified_song`).

We will discuss each widget in detail in the following sections, but here is a brief introduction:

① Add a song to the library

② Load songs from a folder

③ A list of songs in the library. Shows all the songs currently in the library, and also allows the user to select a song for examination and modification

④ A search bar to filter songs

⑤ Shows information about the selected song

⑥ A spin button to adjust time scaling (i.e., pixel width per time unit) of the song visualizations

⑦ Shows a visualization of the selected song

⑧ Shows a visualization of the song after modification

⑨ A spin button to modify song speed

⑩ A spin button to modify song octave

⑪ A drop-down list to select from pre-defined instrument mapping tables

⑫ A drop-down list to select from pre-defined note mapping tables

⑬ Saves the modified song back to disk

⑭ Removes the song from the library

The user can use buttons ① and ② to add songs to the library. Once songs are loaded into the library, they will be shown in ③. Figure 2a shows what it looks like when a user is interacting with the GUI after songs have been loaded into the library.

Once a song is selected, information about the song will be shown in ⑤. ⑦ shows a visualization of the selected song in which each line represents a note. The horizontal axis shows the overall timeline, such that notes beginning at a given time start at the same horizontal position, and notes ending at a given time terminate at the same horizontal position. It should be noted that this visualiztion of absolute event timings is at odds with the delta-time format of midi files. This is something you will need to consider in your implementation. Additionally, this axis may be controlled to some extent by the spin buttons ⑥ and ⑨, which can have a stretching or shrinking effect on the timeline.

The vertical axis of the visualization corresponds to the pitch of the note, which may be impacted by ⑩ and ⑫. Additionally, notes are color-coded by instrument value, which can be altered via ⑪. Figure 2b also shows the GUI in action with some such modifications being performed. Note the thin black line bisecting the visualizations. This line represents middle C (C4, or pitch 60), and serves to provide more context for the octave shift operation. Before applying any modifications to the song, you will need to make a deep copy of `g_current_node->song` and save it in `g_modified_song` so that your modification will not be reflected in the original song until the user clicks "Save Song" ⑬. Lastly, ⑭ removes the song from the internal library, a change that should be reflected in ③.

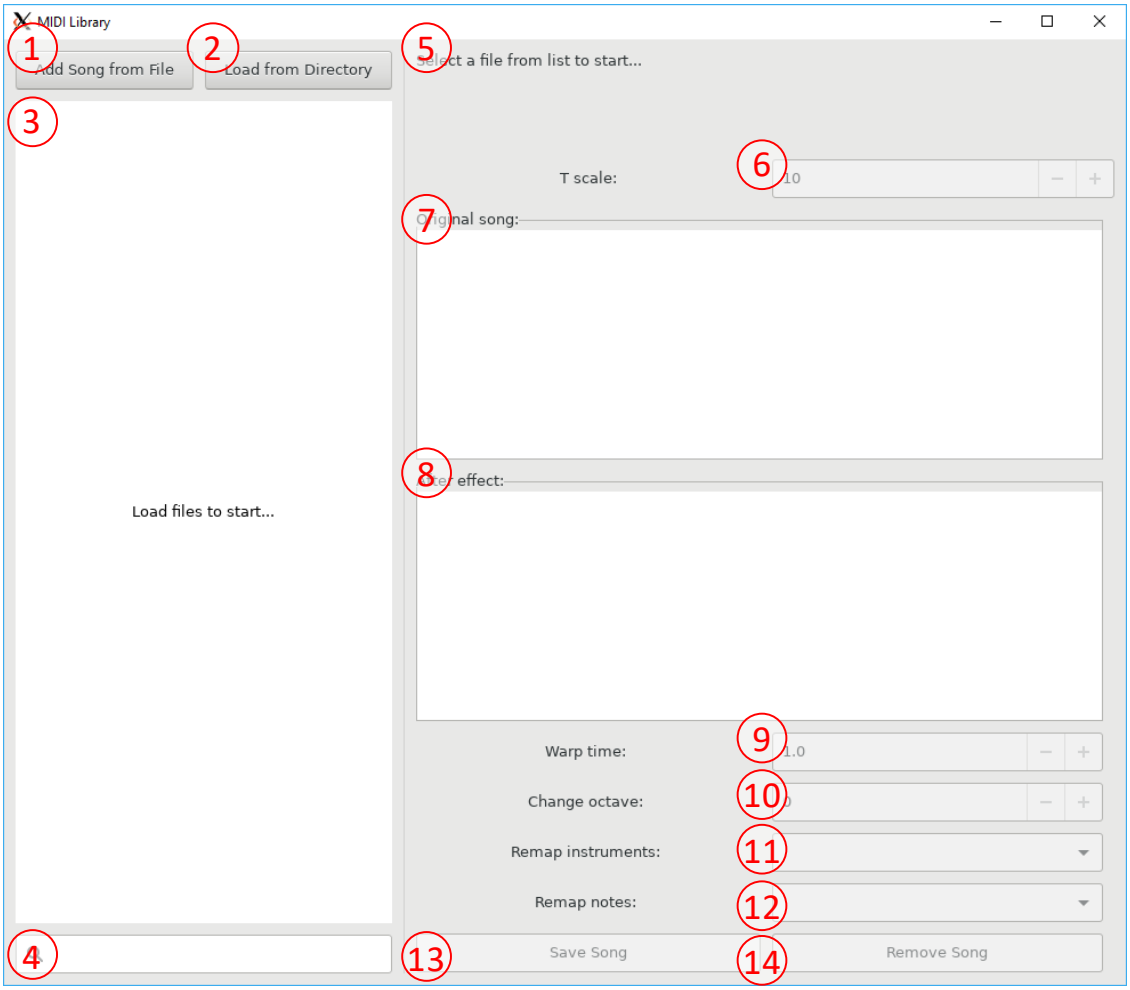
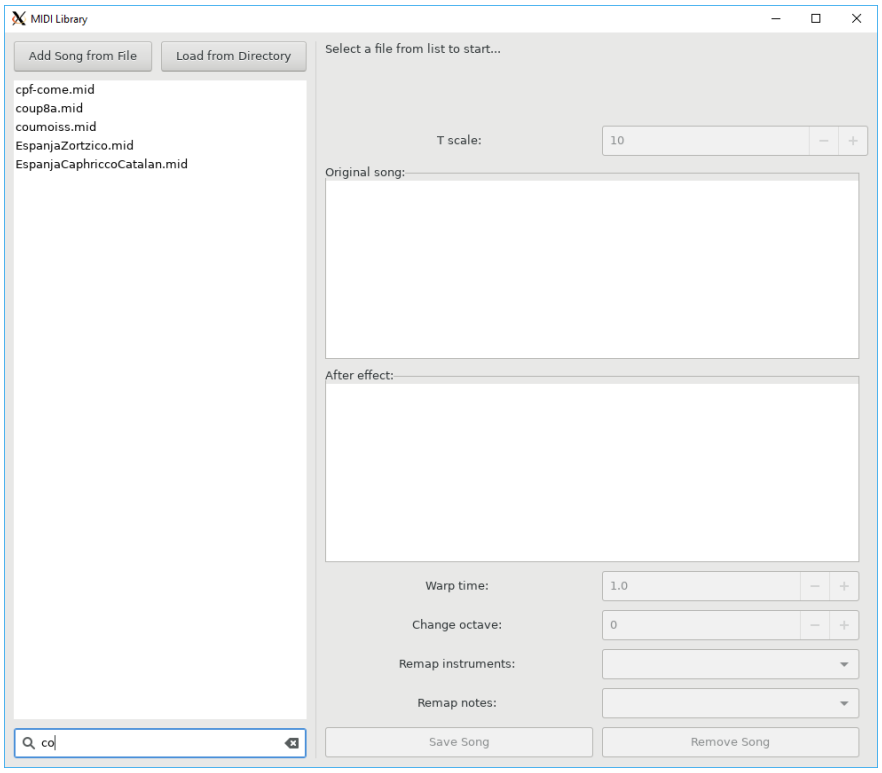
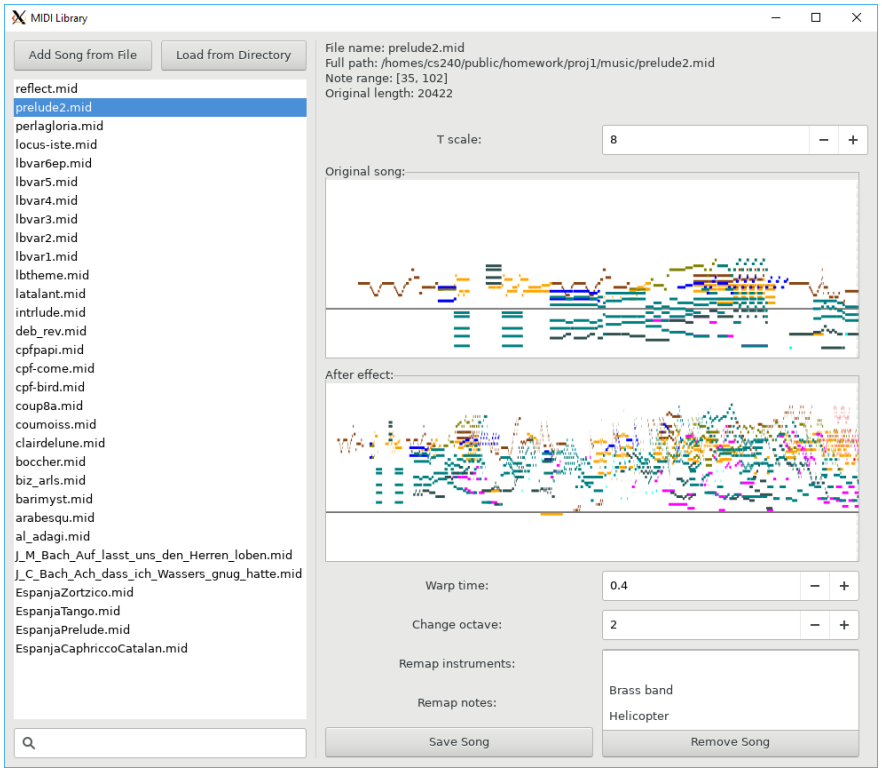


Figure 1: An example UI of the song library.



(a) Filtering the song list for names containing `co`



(b) Modifying a selected song

Figure 2: User interaction with the application.

4.4.2 Functions to Implement

Helper functions

Some of the UI widgets need to be updated frequently and/or in a number of circumstances. The below helper functions should help you update everything in a consistent manner.

`void update_song_list();`

Updates the song list (③) according to the content in `g_song_library`.

`void update_drawing_area();`

Redraws song visualizations in ⑦ and ⑧.

`void update_info();`

Updates the status of ⑤–⑭ according to `g_current_node`. More specifically, updates the text in ⑤, updates the contents in ⑦ and ⑧, and enables/disables ⑥ and ⑨–⑭ depending whether there is a song selected or not. Figure 2a shows an example when no song is selected, and Figure 2b shows an example when a song is selected. Pay attention to the different statuses of ⑥ and ⑨–⑭.

`void update_song();`

Applies the modifications specified in ⑨–⑭ to `g_current_node` and saves the result in `g_modified_song`. Also updates ⑧.

`void range_of_song(song_data_t *, int *, int *, int *);`

This function takes in a song and calculates the range of pitches in the song, as well as the song's length. To avoid messy return types, the relevant values will instead be written to the addresses pointed to by the input int pointers.

The lowest note pitch in the song should be saved to where the first int parameter points to; the highest should be saved to where the second int parameter points to; and the length of the song, in terms of accumulated delta-time, should be saved to where the last int parameter points to.

Be aware that any of the int pointers can be `NULL`, in which case the caller does not need that piece of information, so you are free to ignore it.

UI functions

As mentioned above, GTK is event-driven, and so uses callback functions to handle user interactions. Here are some UI related functions you will need to implement.

`void activate(GtkApplication *, gpointer);`

Function to be called to when application starts. Constructs all widgets and puts them in the window in a manner that is similar to that in Figure 1. It should also connect the events of each widget to corresponding callback functions, then show the window.

The user may want to add a lot of songs to the library, so make sure the user can scroll the song list (③) up and down (vertically). Similarly, ⑦ and ⑧ can get very wide, so make sure to allow the user to scroll them horizontally.

The size of the drawing areas (⑦ and ⑧) should be no smaller than 600×200 pixels so that the song visualizations are easy to see.

Each of the spin buttons should have a specific range: ⑥ should allow the user to select a number from 1 to 10000 inclusive with step of 1; ⑨ should allow the user to select a number from 0.1 to 10.0 inclusive with step of 0.1; and ⑩ should allow the user to select a number from -5 to 5 inclusive with step of 1.

⑪ should allow the user to select a remapping table from: no remapping, and the pre-defined remapping tables `I_BRASS_BAND` and `I_HELICOPTER`. ⑫ should allow the user to select a remapping table from: no remapping, and the pre-defined remapping table `N_LOWER`.

The layout of your implementation does not need to be exactly the same as that in Figure 1; feel free to use your imagination to make a much fancier UI. Just make sure you have all of the widgets mentioned and that they function according to the specifications.

```
void add_song_cb(GtkButton *, gpointer);
```

Callback function of the `clicked` event of the "Add Song" button ①, i.e. when the button is clicked. Creates a dialog to allow users to select a single file. After the file is selected, add that file's name to the library `g_song_library` and update the song list ③ accordingly. You may assume the chosen file is always a valid MIDI file.

```
void load_songs_cb(GtkButton *, gpointer);
```

Callback function of the `clicked` event of the "Load Songs" button ②. Creates a dialog to choose a folder. Adds each MIDI file (i.e. file with extension ".mid") in the selected folder to the library `g_song_library` and updates the song list ③ accordingly.

```
void song_selected_cb(GtkListBox *, GtkListBoxRow *);
```

Callback function of the `row-activated` event of the song list ③, i.e. when a row in the list is selected (clicked). Retrieves the song name from the selected row, locates the tree node in the library which contains the song, and updates the pointer `g_current_node` to point to that tree node. The information about this song should also be shown in ⑤ in the following format (also demonstrated in Figure 2b):

```
File name: [name]
Full path: [path]
Note range: [[lowest_note], [highest_note]]
Original length: [length]
```

```
void search_bar_cb(GtkSearchBar *, gpointer);
```

Callback function of the `search_changed` event of the search bar ④, i.e. when the text in the search bar is changed by user input. Update the song list ③ according to the text in the search bar so that it only shows the songs in the library that contains the keyword (case-insensitive).

Note: this function should not update `g_song_library`, but should just show the filtered results in the list ③.

`void time_scale_cb(GtkSpinButton *, gpointer);`

Callback function of the `value-changed` event of the time scale spinner ⑥, i.e. when the user changes the value using the spinner. Adjust the scaling factor (i.e., width per time unit) along horizontal axis of both ⑦ and ⑧. For example, if the value is 10—as shown in Figure 2b—it means that each pixel in the drawing area represents 10 units of time-delta in the MIDI file. See also `draw_cb`.

`gboolean draw_cb(GtkDrawingArea *, cairo_t *, gpointer);`

Callback function of the `draw` event of the drawing areas (⑦ and ⑧). Both drawing areas share the same callback function, since they are doing mostly the same job, i.e. reading data from a `song_data_t *` and rendering it to the screen. This function draws lines for each of the notes in the song using different colors. Each color represents a different instrument. Before implementing this function, think about how to determine which song data you should use for rendering.

To start, you first need to find out the range of notes and the length of the song. The length of the song determines the width of the drawing area (recall that you should allow the user to scroll horizontally!). The time scaling factor value from ⑥ determines the scaling along the horizontal direction, and the range of notes determines scaling along vertical direction. In particular, the top of the area should be the max among the original and modified songs' highest notes and middle C. Likewise, the bottom of the area should be the minimum among the lowest notes and middle C. Be sure to provide some indication of where middle C lies in both songs that remains visible while scrolling. In Figure 2b, this is manifest as a thin line along the entirety of the visualization windows.

The next step is to traverse the song for all the events in it. Each `Note on` event determines the horizontal starting point of a line, and each `Note off` event corresponds to the horizontal ending point of a line. The vertical location of the line is determined by the pitch of the note. While traversing the `event_list`, you need to keep track of two things: the absolute time, since each event only contains a `delta_time`; and all the notes that have begun but not yet ended, as most music involves playing more than one note at a time.

Additionally, a MIDI file can contain notes played by multiple instruments. To help visualize this, we have provided a mapping table (`COLOR_PALETTE`) from instrument type to line color.

Note: A `Note on` event with a speed of 0 is also treated as a `Note off`.

`void warp_time_cb(GtkSpinButton *, gpointer);`

Callback function of the `value-changed` event of the song speed spinner ⑨. Adjusts the song speed and updates the drawing area ⑧ to reflect the result after the modification is applied.

`void song_octave_cb(GtkSpinButton *, gpointer);`

Callback function of `value-changed` event of the song octave spinner ⑩. Adjusts the song octave and updates the drawing area ⑧.

`void instrument_map_cb(GtkComboBoxText *, gpointer);`

Callback function of the `changed` event of the instrument remapping drop down list ⑪. Applies the chosen instrument remapping and updates the drawing area ⑧.

`void note_map_cb(GtkComboBoxText *, gpointer);`

Callback function of the `changed` event of the note remapping octave spinner ⑫. Applies the chosen note remapping and updates the drawing area ⑧.

`void save_song_cb(GtkButton *, gpointer);`

Callback function of the `clicked` event of the "Save Song" button ⑬). Saves the modified song `g_modified_song` back to the file it originally came from. Also updates `g_current_node` to ensure it contains the updated song.

Note: you will not be able to test this function on the shared music directory provided to you, for obvious reasons. Accordingly, we recommend downloading a copy of the music directory to your local machine and running your tests there.

`void remove_song_cb(GtkButton *, gpointer);`

Callback function of the `clicked` event of the "Remove Song" button (⑭). Removes the selected song `g_current_node` from `g_song_library` (but does not delete the song from the disk!). Don't forget to update all the related UI widgets. All related memory should be freed as well.

How to start

- As always, start early!
- Start from the simplest example code online and get a high level idea of how it works.
- We've included some sample code in ui.c which creates a window with a text label and two buttons. Try to understand how it works first.
- Using both online and local examples as references, write your own code. Start with minimal modifications of the examples, then try something bigger.

4.5 Input Files

For this project, you will need access to a directory structure containing MIDI files. To avoid angering the system administrators, we will not be distributing 400+ copies of the same music library to all students. Instead, we have exposed ~cs240/public/homework/proj1/music with read-only access. Feel free to use this directory structure and the files therein for testing.

We may periodically update this directory structure with more music, so you shouldn't make any assumptions about its contents in your code.

4.6 Header Files

We provide a number of header files for you in this assignment. You should not alter them file, as we will replace them with the originals when grading. That said, feel free to implement any number of helper functions in your .c files.

These Standard Rules Apply

- You may add any `#includes` you need to the top of your *.c files.
- You may not create any global variables other than those that are provided for you **or** suggested in the instructions. Creation of additional global variables will impact your style grade.
- You should check for any failures and return an appropriate value.
- You should not assume any maximum size for the input files. The test program will generate files of arbitrary size.
- Do not look at anyone else's source code. Do not work with any other students.

Submission

This project will be divided into 3 checkpoints which must be submitted independently by their stated due dates.

To submit the relevant checkpoint for grading, type:

```
$ make submit_checkpoint_1
```

Or

```
$ make submit_checkpoint_2
```

```
$ make submit_checkpoint_3
```

In your proj1 directory. You can do this as often as you wish. We encourage you to submit your code as often as possible. Only your final submission will be graded.

5 Grading

Each checkpoint will be worth 50 points. Specific test cases may be released partway through each phase, but we reserve the right to withhold some of the test cases. In general, you will need to verify by yourself that your program functions correctly.

In addition, your code will be assessed according to the code standard, which will offer up to a 5% deduction in your overall, cross-checkpoint score.

Further details or alterations may be released via Piazza as time progresses.

Your code must compile successfully using `-Wall -Werror -std=c99` to receive any credit. Code that does not compile will be assigned an automatic score of 0.

If your program crashes (e.g., segmentation fault) at any point during the testing process, your score will be reduced by 5% of the available points for each checkpoint

If your program exhibits any memory errors (e.g., corruption, double free) at any point during the testing process, your score will also be reduced by 5% of the available points for each checkpoint