# CS302 -- Lab 3 -- Manipulating MIDI

- **CS302 -- Fundamental Algorithms**
- **Spring, 2022**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab3**
- **Lab Directory: /home/jplank/cs302/Labs/Lab3**

## This Long Lab Writeup

This is a very long lab writeup. We will go over it in class or lab. Please read it *thoroughly*, and play around with hand-editing the files before you start writing code.

---

## What you submit

You will submit your **cs302-midi.cpp** file. *You may not modify **cs302-midi-base.cpp** or **cs302-midi.h** in any way.* The TA will compile your **cs302-midi.cpp** with my **cs302-midi-base.cpp** and **cs302-midi.h**.

There is a makefile in the lab directory. If you copy **cs302-midi.cpp**, **cs302-midi-base.cpp** and **cs302-midi.h** and the **makefile** from the lab directory, then you should be able to compile with **make**, and then start editing **cs302-midi.cpp** to do the lab.

---

## Introduction: MIDI

MIDI stands for "*Musical Instrument Digital Interface*". It is a bit of an overloaded term, so I'll try to give it a succinct explanation here. If you want more, there's tons of text about MIDI on Wikipedia.

MIDI is a standard for representing and manipulating note-based music on computers. Most electronic instruments, notably keyboards, have MIDI output ports which emit events in a specific format whenever the instruments are played. These events are emitted in real-time, so that a computer may record them, manipulate them, etc.

There is a file format called *Standard MIDI File (SMF) Format*, which records a MIDI performance. These files contain MIDI events that an instrument would emit, plus timing information about when the various events occur. A MIDI player will play these events back, using some sort of sound synthesizer.

Most web browsers support SMF files. For example, **bach_565.mid** contains a rather tepid performance of Bach's famous D minor Toccata and Fugue (BWV 565) (taken from **http://users1.ee.net/lstone/midi.htm**). The file **rockytop.mid** contains a more familiar tune. When you click on these links, your web browser should play the MIDI files. If your computer has a nice expensive sound card, this will sound pretty good. If it has a janky one like mine, it will sound kinda lame. So it goes.

The SMF format is far too detailed for this lab. Instead, we're only going to concern ourselves with a limited subset of SMF, and we'll use two different representations that are easy to edit and manipulate. MIDI files contain *tracks*, each of which is a different instrument. A MIDI player will play all of the tracks simultaneously, which can make for a good performance. We are only going to concentrate on MIDI files with a single track.

These MIDI files contain linear streams of *events*. We will only handle four of these events:

- **ON**: Turn a note on. The parameters of this event are the pitch of the note and its volume, each of which is an integer between 1 and 127.
- **OFF**: Turn a note off. The parameter of this event is the pitch of the note. A note should be turned on before it is turned off.
- **DAMPER DOWN**: Depress the damper pedal. The damper pedal on a piano makes sure that all notes that are playing keep playing, regardless of whether they are turned OFF, until the damper pedal is released.
- **DAMPER UP**: Release the damper pedal.

We are going to handle two representations of MIDI files. You may read their specifications in the following links:

1. Event Files: In these you specify events in a linear stream, where each event contains the duration between it and the previous event.

2. Note Files: In these, you specify note events and damper pedal events. A note event is represented by a pitch, volume, start time and stop time. Thus, it stores two MIDI events -- an **ON** and an **OFF** event. A damper pedal event is represented by a start and stop time. Like a note, it stores two MIDI events -- a **DAMPER DOWN** and a **DAMPER UP** event.

   Unlike Event files, Note files are not a linear stream of events -- since each event has a start and a stop time, events may be specified in any order.

Again, read the specification files for more information and for simple example files that illustrate their use. You should go ahead and hand-edit some of these files and convert them to MIDI to reinforce your understanding of MIDI.

The program **mconvert** converts Midi-Event Files to and from Midi-Note Files:

```
mconvert inputfile outputfile E|N
```

The **inputfile** can be either a Midi-Event-File or a Midi-Note-File. The format of the **outputfile** is either an Event file (if the last argument is **E**) or a Note file (if the last argument is **N**).

---

# cs302-midi.h

Take a look at **cs302-midi.h**

```
#include <iostream>
#include <map>
#include <list>
using namespace std;

class ND {
  public:
    int key;          // 'N' for Note, or 'D' for Damper
    int pitch;        // Ignored by 'D'
    int volume;       // Ignored by 'D'
    double start;
    double stop;
};

class Event {
   public:
     int key;         // 'O':ON, 'F':OFF, 'D':DAMPER,
     int time;
```

```
    int v1;          // Pitch for O/F, 1 for D:DOWN, 0 for D:UP
    int v2;          // Volume for O.  Ignored for everything else.
};

typedef multimap <double, ND *> NDMap;
typedef list <Event *> EventList;

class CS302_Midi {
  public:
    CS302_Midi(string file);
    ~CS302_Midi();
    void Write(string file, char format);   // 'E' for Event, 'N' for Note
    void Add_Pause(double starttime, double duration);
    void Scale(double factor);
  protected:
    void destroy_nd();
    void destroy_el();
    void nd_to_el();
    void el_to_nd();
    EventList *el;
    NDMap *nd;
};
```

This defines a class called **CS302_Midi**, which lets you read, manipulate and write both representations of Midi files. The constructor takes a filename and creates an instance of the class from the file. The file can be in either format (the first word in the file defines the format). There is a destructor method, which is necessary because the constructor makes **new** calls. Additionally, there is a **Write()** method, which writes out the class instance in either format.

There are two other public functions -- **Add_Pause()** adds a pause to the file at the given start time and duration (both in seconds). **Scale()** scales the speed of the file -- for example, scaling by 2 will make it play twice as fast, and scaling by .5 will make it play half as fast.

There are four protected methods and two protected variables. Let's start with the variables:

- **el**: This is a list of events as defined in a Midi-Event-File. As you can see from the **EventList** typedef, it is a list of pointers to **Event** class instances. When you create an instance of **CS302_Midi** from a Midi-Event-File, it creates **el** directly from the file. Each **Event** in the event list is categorized by its **key**, which is '**O**' for an **ON** event, '**F**' for an **OFF** event, and '**D**' for either a **DAMPER DOWN** or **DAMPER UP** event. The **time** is the time (in 1/480 of a second) since the previous event. **v1** depends on the **key** -- if **key** is '**O**' or '**F**', then **v1** is the pitch. Otherwise, it is 1 for **DAMPER DOWN** and 0 for **DAMPER UP**. **v2** is the volume of an '**O**' event. It is ignored for all other events.

- **nd**: This is a multimap of NOTE and DAMPER events as defined in a Midi-Note-File. Each of these events has a start time, and the multimap is keyed on these start times. Each multimap entry points to a **ND** class instance. This contains all the information about the NOTE or DAMPER event instances. When you create an instance of **CS302_Midi** from a Midi-Note-File, it creates **nd** directly from the file.

  Like an **Event**, a **ND** instance has a **key**, which is '**N**' for a note and '**D**' for a damper pedal event. The rest of the fields should be self-explanatory.

Two of the private methods are straightforward: **destroy_el()** deletes **el**, making sure to delete every **Event** in the list. **destroy_nd()** delete **nd**, making sure to delete every **ND** in the multimap. These are called by the destructor, and also by **Add_Pause()** and **Scale()**.

The last two methods are the tricky methods: **el_to_nd()** creates the multimap **nd** from the event list **el**. Conversely, **nd_to_el()** creates the event list **el** from the multimap **nd**. These are called by the constructor -- if the constructor reads a Midi-Event-File, it creates **el** from the file and then creates **nd** using **el_to_nd()**.

Conversely, if the constructor reads a Midi-Note-File, it creates **nd** from the file and then creates **el** using **nd_to_el()**.

Thus, when the constructor is done, both **el** and **nd** are initialized to represent the same MIDI file. This makes it easy to write **Write()** -- it uses **el** to create the Midi-Event-File output and it uses **nd** to create the Midi-Note-File output.

---

## Your Job

The entire **CS302_Midi** class has been implemented for you, with the exception of these last two methods. Additionally, there are two programs **mconvert.cpp** and **midi_tester.cpp**, which test the implementation.

Your job is to implement the two unimplemented methods.

The implementations that I provide for you are in **cs302-midi-base.cpp**. The remaining two methods are given a skeleton implementation in **cs302-midi.cpp**. This means that you can copy them to your directory along with the **makefile** and they will compile:

```
UNIX> make clean
rm -f *.o midi_tester mconvert  core
UNIX> make
g++ -c midi_tester.cpp
g++ -c cs302-midi-base.cpp
g++ -c cs302-midi.cpp
g++ -o midi_tester midi_tester.o cs302-midi-base.o cs302-midi.o
g++ -c mconvert.cpp
g++ -o mconvert mconvert.o cs302-midi-base.o cs302-midi.o
UNIX>
```

First, let's consider **mconvert**. This reads a file and then writes a file. If we write a file of the same type, this version will work. For example:

```
UNIX> mconvert C-Major-MEF.txt tmp.txt E
UNIX> cat tmp.txt
CS302-Midi-Event-File
ON 0 60 64
ON 0 64 64
ON 0 67 64
OFF 480 60
OFF 0 64
OFF 0 67
UNIX> mconvert C-Major-MNF.txt tmp.txt N
UNIX> cat tmp.txt
CS302-Midi-Note-File
NOTE 60 64 0 1
NOTE 64 64 0 1
NOTE 67 64 0 1
UNIX>
```

The first call works because reading **C-Major-MEF.txt** reads all the events into **el** in the constructor. The constructor also calls **el_to_nd()**, but that doesn't do anything. However, when we write the file with format **E**, it write out **el**, which works fine.

Similarly, when we read **C-Major-MNF.txt**, the constructor creates the **nd** multimap. It also calls **nd_to_el()**, but that doesn't do anything. When we print using format **N**, that traverses **nd** and prints out all the notes.

If we try to read a MEF file and print it with format **N**, the resulting file will be empty, because **el_to_nd()** has no implementation. Similarly, if we try to read a MNF file and print it with format **E**, the resulting file will be empty:

```
UNIX> mconvert C-Major-MEF.txt tmp.txt N
UNIX> cat tmp.txt
CS302-Midi-Note-File
UNIX> mconvert C-Major-MNF.txt tmp.txt E
UNIX> cat tmp.txt
CS302-Midi-Event-File
UNIX>
```

At this point, make sure you have understood everything. Make sure you try the above examples, and perhaps some others. Read the implementation of the **Write()** method and the constructor method so that you know how **el** and **nd** work.

---

## Your first implementation: nd_to_el()

First, implement **nd_to_el()**. It is the easier of the two methods. It should traverse the **nd** multimap and create **ON/OFF/DAMPER DOWN/DAMPER UP/** events (of type **Event ***) from them. Think about how you should implement this. You'll need some temporary data structures. You also need to worry about what happens when multiple events occur at the same time. For example, look at **Two-Repeat-MNF.txt**:

```
CS302-Midi-Note-File
NOTE 60 65 0 1
NOTE 60 60 1 2
```

The first note ends at time 1, and the second note begins at time 1. You would not want this file to convert to the following MEF file:

```
CS302-Midi-Event-File
ON 0 60 65
ON 480 60 60
OFF 0 60
OFF 480 60
```

Why? Because you would not have an OFF event corresponding to the first ON event until the note is played a second time. Instead, you want **Two-Repeat-MNF.txt** to convert to:

```
CS302-Midi-Event-File
ON 0 60 65
OFF 480 60
ON 0 60 60
OFF 480 60
```

So, here is what you should do in **nd_to_el()**:

- Convert each '**N**' event into an **ON** and an **OFF** event. Also convert each '**D**' event into a **DAMPER DOWN** and a **DAMPER UP** event.
- Insert each of these events into a temporary map keyed on the start time, which are integers whose units are 1/480 of a second. Use **rint()** to round the double to an integer.

  You are going to insert these events into a map and not a multimap. The val part of the map should be a multimap containing all of the events that start at that time. The events should be in the following relative order:

  - **OFF** events.
  - **DAMPER UP** events.

- **DAMPER DOWN** events.
- **ON** events.

That way, notes will be turned off before they are turned on, and the damper pedal will be set to up before it is set to down.

One final part of **nd_to_el()** -- if a **'N'** or **'D'** event is such that it will start and stop at the same time once you convert them to integers, ignore the event. For example, the following note should be ignored when you call **nd_to_el()** because **rint(0.00000001*480.0) = rint((0.00000002*480.0) = 0**:

```
NOTE 60 65 0.00000001 0.00000002
```

- Traverse the temporary map and push all the events onto **el()**, calculating the correct time for each event.

---

## Your second implementation: el_to_nd()

The method **el_to_nd()** is harder. You are going to need a vector of 128 **(ND *)**'s. Initialize them all to NULL. Then, when you see an **ON** event, you will create a **ND** for it, set its start time and put it into the vector, indexed by pitch. When you see an **OFF** event, you will use the **ND** in the vector, set its stop time, and insert it into **nd**. Similarly, you should have a **(ND *)** for the damper pedal, which you create when you encounter a **DAMPER DOWN** event, and which you finish and insert into the map when you encounter a **DAMPER UP** event. At the end of traversing **el**, you should double-check the vector and the damper variable and make sure that there are no notes that have **ON** events that are not finished with corresponding **OFF** events.

There is some additional error checking that you should do. I will not grade you on it. However, you should be able to catch the following errors when you write **el_to_nd()**:

- An **OFF** event for a pitch that is not preceded by an **ON** event.
- An **ON** event for a pitch that follows another **ON** event.
- An **ON** event for a pitch that is never followed by an **OFF** event.
- A **DAMPER UP** event that is not preceded by a **DAMPER DOWN** event.
- A **DAMPER DOWN** event that follows another **DAMPER DOWN** event.
- A **DAMPER DOWN** event that is never followed by a **DAMPER UP** event.

When you are done with **el_to_nd()** and **nd_to_el()** the programs **mconvert** and **midi_tester** should work with complete functionality.

---

## Files

There are working executables for **mconvert** and **midi_tester**, as well as a **cs302-midi.o** that contains working implementations of **el_to_nd()** and **nd_to_el()**. When it doubt, make sure that your programs work like these. Again, I will not grade you on how your programs error check, so don't stress about the output format.

Also, the gradescript makes use of two programs: **event-file-grader** and **note-file-grader**. These take two Event and Note files respectively and print out if they differ. The **note-file-grader** ignores the order of note specifications in the input files -- it simply makes sure that both files have the same notes.

The **event-file-grader** is a little more detailed. It partitions the events into "on" events (NOTE-ON and DAMPER DOWN) and "off" events (NOTE-OFF and DAMPER UP). If events of the same type occur simultaneously, then they may occur in any order. For example, the following Midi-Event-Files are equivalent, although not identical, because:

| | |
|---|---|
| DAMPER 5 DOWN<br>ON 0 60 65<br>ON 0 54 55<br><br>OFF 480 60<br>OFF 0 54<br>DAMPER 0 UP | ON 5 54 55<br>ON 0 60 65<br>DAMPER 0 DOWN<br><br>OFF 480 54<br>DAMPER 0 UP<br>OFF 0 60 |

However, the following are not, since you can't reorder "on" and "off" events relative to each other, even though they happen at the "same time:"

| | |
|---|---|
| ON 0 60 65<br><br>DAMPER 5 DOWN<br>OFF 0 60<br><br>DAMPER 20 UP | ON 0 60 65<br><br>OFF 5 60<br>DAMPER 0 DOWN<br><br>DAMPER 20 UP |

---

## If you get inspired

If you get inspired to create some neat MIDI files, I'd love to see them -- when you're done, feel free to email me MEF or MNF files that you have created. If I like them, I'll post them with the lab for future years to see.

---

## A Final Word

I know that this is a brutal lab writeup. It is long, and I'm guessing it may take the entire lab session to read and understand. This is part of computer science -- understanding specifications, conventions, and formats, and working within those constraints. Take it slow, and try to understand the lab material piece by piece. It should be a rewarding lab!

---

## Piazza Q & A

- 2022-02-15-Question.pdf