Lab 3 - lab.h

```
#ifndef LAB_H
# define LAB_H

# include <iostream>
# include <fstream>
# include <catch.hpp>
# include <vector>
# include <string>
# include <cstring>
# include <abc.h>

# define MAX_FRAGMENT 255

using namespace std;

extern double    TEMPO;

struct            Note
{
    string        tone;
    float        duration;
    string        frag;
};

struct            Fragment
{
    int            st;
    int            end;
};

struct            MyStack
{
    int            size;
    Fragment    *buf;
    int            sp;
};
```

Libraries:
iostream: for for stdin/stdout via terminal
fstream: for read/write file
vector: for vector
abc: ABC to SOX associative table

Structs:
Note: for the each note
Fragment: for the stack element
MyStack: stack

Global:
TEMPO: for the tempo of the song

Function Declaration

```
37 /*
38 **        ABC Play functions
39 */
40
41 bool    readABCfile(string fileName, vector<string> &v);
42 void    playSong(const vector<Note> &soxVector);
43 void    convertABCtoSOX(const vector<string> &abc, vector<Note> &sox);
44 string    createPlayCmd(const Note &n);
45
46 /*
47 **        Stack functions
48 */
49
50 bool    myCreate(MyStack &stack, size_t size);
51 bool    myPush(MyStack &stack, Fragment item);
52 bool    myPop(MyStack &stack, Fragment &item);
53 void    myDestroy(MyStack &stack);
54
55 #endif
```

```
#ifndef ABC_H
# define ABC_H

/*
**    <Associative array>
**
**    Examples:
**    ABC: C, C  c  c' | C2    | C/2
**    SOX: C3 C4 C5 C6 | 2 sec | half sec
**
**    You can get sox notation following:
**    sox = abcToSox[abc];
**    it will get the corresponding notation
*/

# include <map>

typedef std::map<std::string, std::string> abc_t;
extern abc_t    abcToSox;
extern abc_t    abcToSox_fragment;

#endif
```

ABC header:
This is a custom header and cpp file for
translating ABC notation to SOX notation.
There is an associative table that
converts ABC notation to SOX notation.

```
#include <lab.h>

abc_t    abcToSox_fragment = {
    {"|",  "MEASURE"},
    {"|:", "REPEAT_BEGIN"},
    {":|", "REPEAT_END"},
    {"::", "REPEAT_BOTH"},
    {"||", "STOP"}
};
```

ABC:
The associative table is declared here
(only for measure notations), and
it will be exported as a global variable.

Lab 3 - readABCfile.cpp

```cpp
#include <lab.h>

double    TEMPO = 0.25;
void    getTempo(string s);

bool    readABCfile(string fileName, vector<string> &v)
{
    ifstream    ifs(fileName);
    bool        ret = true, k = false;

    if (ifs)
    {
        string    s;

        while (!k)
        {
            getline(ifs, s);
            if (s[0] == 'L')
                getTempo(s);
            if (s == "K:C")
                k = true;
        }
        while (ifs >> s)
            v.push_back(s);
    }
    else
        ret = false;
    return ret;
}

void    getTempo(string s)
{
    double    n;
    double    d;

    n = atoi(strchr(s.c_str(), ':') + 1);
    d = atoi(strchr(s.c_str(), '/') + 1);
    n /= d;
    TEMPO /= n;
}
```

readABCfile:
This function opens the file and reads the data.
Then, store the data into a vector of strings.
It will skip the information part in the beginning,
only store L:, which is the tempo.

getTempo:
This function calculates the tempo of the song
based on 1/4 is tempo 1.

```cpp
#include <lab.h>

void    convertABCtoSOX(const vector<string> &abc, vector<Note> &sox)
{
    for (size_t i = 0; i < abc.size(); i++)
    {
        if (abcToSox_fragment.find(abc[i]) != abcToSox_fragment.end())
        {
            sox[i].tone = "";
            sox[i].duration = 0;
            sox[i].frag = abcToSox_fragment[abc[i]];
            continue ;
        }
        size_t j = 0;
        if ((abc[i][j] >= 'A' && abc[i][j] <= 'G') ||
            (abc[i][j] >= 'a' && abc[i][j] <= 'g'))
        {
            sox[i].tone += abc[i][j++];
            if (abc[i][j] == ',' || abc[i][j] == '\'')
            {
                sox[i].tone += (abc[i][j] == ',') ? "3" : " 6";
                j++;
            }
            else
                sox[i].tone += sox[i].tone[0] < 97 ? "4" : "5";
            if (abc[i][j] == '/')
                sox[i].duration = 1.0 / atoi(&abc[i][j + 1]);
            else
                sox[i].duration = atoi(&abc[i][j]);
            sox[i].duration == 0 ? sox[i].duration = 1 : 0;
            sox[i].tone[0] &= -33;
            sox[i].frag = "";
        }
    }
}
```

convertABCtoSOX:
This function checks the ABC notation and covert it into the SOX notation.
Also, it gets the duration and fragments.

It checks every letter in the string and gets the note.
For the duration, gets the number using atoi.
At the end it changes lower case to upper case using AND bit operator.

Lab 3 - createPlayCmd.cpp

```cpp
#include <lab.h>

string    createPlayCmd(const Note &n)
{
    return
    (
        "play -n synth " + to_string(n.duration) + " pluck " +
        n.tone + " tempo " + to_string(TEMPO)
    );
}
```

createPlayCmd:
This function takes a Note struct and
creates a SOX command. It combines the command
template and the data in the struct.

Lab 3 - playSong.cpp

```cpp
#include <lab.h>

void    playSong(const vector<Note> &soxVector)
{
    MyStack    myStack;
    Fragment frag;
    size_t    curr = 0;
    size_t    tmp;
    size_t    end = soxVector.size() - 1;

    if (!myCreate(myStack, MAX_FRAGMENT))
    {
        cerr << "Failed to create stack." << endl;
        exit(0);
    }
    while (true)
    {
        if (!soxVector[curr].tone.empty() && curr < end)
            system(createPlayCmd(soxVector[curr++]).c_str());
        else if (!soxVector[curr].frag.empty() && curr < end)
        {
            if (soxVector[curr].frag == "REPEAT_BEGIN")
                tmp = ++curr;
            else if ((soxVector[curr].frag == "REPEAT_END") ||
                (soxVector[curr].frag == "REPEAT_BOTH"))
            {
                frag.st = curr + 1;
                frag.end = end;
                end = curr;
                curr = tmp;
                myPush(myStack, frag);
                if (soxVector[frag.st - 1].frag == "REPEAT_BOTH")
                    tmp = frag.st;
            }
            else
                curr++;
        }
        else if (curr == end)
        {
            Fragment    frag_tmp;

            if (myPop(myStack, frag_tmp))
```

playSong:
This function takes the soxVector and plays the song.
First, it creates a new stack to store the indices.
If it meets repeat fragment while it plays the song,
it will push the current index into the stack.
Then, it plays the repeat part again and comes back to the stored index using pop.
When the song ends, it destroy the stack.

```
43              {
44                  curr = frag_tmp.st;
45                  end = frag_tmp.end;
46              }
47              else
48                  break ;
49          }
50      }
51  }
```

```cpp
#include <lab.h>

bool    myCreate(MyStack &stack, size_t size)
{
    stack.buf = new Fragment[size];
    if (!stack.buf)
        return (false);
    stack.size = size;
    stack.sp = 0;
    return (true);
}

#ifdef UNIT_TEST

TEST_CASE("Stack Create")
{
    MyStack stack;

    REQUIRE(myCreate(stack, 10));
    myDestroy(stack);

    REQUIRE(myCreate(stack, 1));
    myDestroy(stack);

    REQUIRE(myCreate(stack, 100));
    myDestroy(stack);

    REQUIRE(myCreate(stack, 1000));
    myDestroy(stack);
}

#endif
```

myCreate:
It creates a new stack.
If allocation fails, returns false.

UNIT_TEST:
I created several times with different sizes.
Then I checked if it returns true.
I couldn't test failure since I don't know
how to make 'new' fail.

Lab 3 - myPush.cpp

```
#include <lab.h>

bool    myPush(MyStack &stack, Fragment item)
{
    if (stack.size == stack.sp)
        return (false);
    stack.buf[stack.sp++] = item;
    return (true);
}

#ifdef UNIT_TEST

TEST_CASE("Stack Push")
{
    MyStack     stack;
    Fragment    item;

    myCreate(stack, 2);

    item.st = 10; item.end = 15;
    REQUIRE(myPush(stack, item));
    REQUIRE((stack.buf[0].st == 10 && stack.buf[0].end == 15));

    item.st = 20; item.end = 25;
    REQUIRE(myPush(stack, item));
    REQUIRE((stack.buf[1].st == 20 && stack.buf[1].end == 25));

    REQUIRE_FALSE(myPush(stack, item));
    myDestroy(stack);
}

#endif
```

myPush:
It pushes the item into the stack.
If the stack is full, returns false.

UNIT_TEST:
I created a new stack with size 2.
Then, I push something twice and check if
the stack has the items in the proper order.
Also, I pushed one more time so that
I can check the failure when it is full.

Lab 3 - myPop.cpp

```
#include <lab.h>

bool    myPop(MyStack &stack, Fragment &item)
{
    if (stack.sp == 0)
        return (false);
    item = stack.buf[--stack.sp];
    return (true);
}

#ifdef UNIT_TEST

TEST_CASE("Stack Pop")
{
    MyStack     stack;
    Fragment    item;
    Fragment    res;

    myCreate(stack, 2);

    item.st = 10; item.end = 15;
    myPush(stack, item);
    item.st = 20; item.end = 25;
    myPush(stack, item);

    REQUIRE(myPop(stack, res));
    REQUIRE((res.st == 20 && res.end == 25));

    REQUIRE(myPop(stack, res));
    REQUIRE((res.st == 10 && res.end == 15));

    REQUIRE_FALSE(myPop(stack, res));
    myDestroy(stack);
}

#endif
```

myPop:
It pops the item in the top of the stack.
If the stack is empty, returns false.

UNIT_TEST:
I pushed 2 items and check if it pops
in the proper order.
After that, I tried to pop the empty stack
and checked if it returns false.

```cpp
#include <lab.h>

void    myDestroy(MyStack &stack)
{
    delete stack.buf;
    stack.buf = NULL;
}

#ifdef UNIT_TEST

TEST_CASE("Stack Destroy")
{
    MyStack    stack;

    myCreate(stack, 10);
    myDestroy(stack);
    REQUIRE_FALSE(stack.buf);
}

#endif
```

myDestroy:
It deletes the stack and set it to null pointer.

UNIT_TEST:
Since delete functions doesn't return any bool expression and the data stays even it was deleted, it is hard to check success/fail of the function. So, I guess it worked if the program doesn't throw an execption error.

CATCH TEST

```
===============================================================
All tests passed (15 assertions in 4 test cases)
```

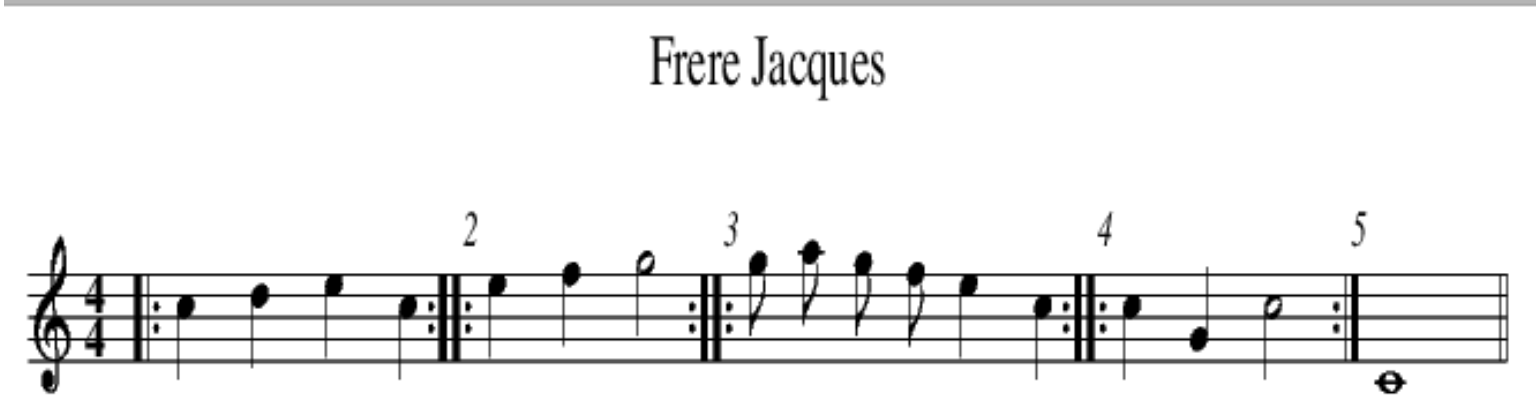
Convert

```cpp
#include <lab.h>

void    convertABCtoSOX(const vector<string> &abc, vector<Note> &sox)
{
    for (size_t i = 0; i < abc.size(); i++)
    {
        cout << "ABC: [" << abc[i] << "]" << endl;
        if (abcToSox_fragment.find(abc[i]) != abcToSox_fragment.end())
        {
            sox[i].tone = "";
            sox[i].duration = 0;
            sox[i].frag = abcToSox_fragment[abc[i]];
            cout << "SOX: [" << sox[i].frag << "] : fragment" << endl;
            continue ;
        }
        size_t j = 0;
        if ((abc[i][j] >= 'A' && abc[i][j] <= 'G') ||
            (abc[i][j] >= 'a' && abc[i][j] <= 'g'))
        {
            sox[i].tone += abc[i][j++];
            if (abc[i][j] == ',' || abc[i][j] == '\'')
            {
                sox[i].tone += (abc[i][j] == ',') ? "3" : " 6";
                j++;
            }
            else
                sox[i].tone += sox[i].tone[0] < 97 ? "4" : "5";
            if (abc[i][j] == '/')
                sox[i].duration = 1.0 / atoi(&abc[i][j + 1]);
            else
                sox[i].duration = atoi(&abc[i][j]);
            sox[i].duration == 0 ? sox[i].duration = 1 : 0;
            sox[i].tone[0] &= -33;
            cout << "SOX: [" << sox[i].tone << "] : tone" << endl;
            sox[i].frag = "";
        }
    }
}
```

```
ABC: [|:]
SOX: [REPEAT_BEGIN] : fragment
ABC: [c]
SOX: [C5] : tone
ABC: [d]
SOX: [D5] : tone
ABC: [e]
SOX: [E5] : tone
ABC: [c]
SOX: [C5] : tone
ABC: [::]
SOX: [REPEAT_BOTH] : fragment
ABC: [e]
SOX: [E5] : tone
ABC: [f]
SOX: [F5] : tone
ABC: [g2]
SOX: [G5] : tone
ABC: [::]
SOX: [REPEAT_BOTH] : fragment
ABC: [g/2]
SOX: [G5] : tone
ABC: [a/2]
SOX: [A5] : tone
ABC: [g/2]
SOX: [G5] : tone
ABC: [f/2]
SOX: [F5] : tone
ABC: [e]
SOX: [E5] : tone
ABC: [c]
SOX: [C5] : tone
ABC: [::]
SOX: [REPEAT_BOTH] : fragment
ABC: [c]
SOX: [C5] : tone
ABC: [G]
SOX: [G4] : tone
ABC: [c2]
SOX: [C5] : tone
ABC: [:|]
SOX: [REPEAT_END] : fragment
ABC: [C4]
SOX: [C4] : tone
ABC: [||]
SOX: [STOP] : fragment
```

Result




```
Done.

  Encoding: n/a
  Channels: 1 @ 32-bit
Samplerate: 48000Hz
Replaygain: off
  Duration: unknown

In:0.00% 00:00:02.05 [00:00:00.00] Out:96.0k [      |      ] Hd:
Done.

  Encoding: n/a
  Channels: 1 @ 32-bit
Samplerate: 48000Hz
Replaygain: off
  Duration: unknown
```