A412

0.1

Generated by Doxygen 1.8.8

Mon Dec 15 2014 08:29:04

Contents

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

2 File Index

2.1 File List

Here is a list of all files with brief descriptions:

3 Data Structure Documentation

3.1 data Struct Reference

Data Fields

- unsigned int **tempo**
- int **ppqn**
- **mode mode**

3.1.1 Detailed Description

A struct containing general data pertaining to the song

Parameters

tempo	the tempo in beats-per-minute
ppqn	ticks-per-quarter-note contains the number of ticks per quarter note
mode	an enumerated value representing the mode (major/minor)

3.1.2 Field Documentation

3.1.2.1 mode data::mode

3.1.2.2 int data::ppqn

3.1.2.3 unsigned int data::tempo

The documentation for this struct was generated from the following file:

- **main.c**

3.2 eventPlacement Struct Reference

Data Fields

- int **noteOn**
- int **noteOff**
- int **afterTouch**
- int **controlChange**
- int **programChange**
- int **channelPressure**
- int **pitchWheel**

3.2.1 Detailed Description

A struct containing placements of midi events, stored as their placement in file

Parameters

noteOn	signals when a note starts playing
noteOff	signals when a note stops playing
afterTouch	changes velocity for a single note on a single channel
controlChange	used for a large number of effects, none of which are used in this project (stored to find deltatimes)
programChange	signals instrument change (not used; stored to find deltatimes)
channelPressure	changes velocity for all notes on a specific channel (akin to a global afterTouch)
pitchWheel	fine tuning of pitch for all notes on a specific channel (similar to channelPressure, but for pitch)

3.2.2 Field Documentation

3.2.2.1 int eventPlacement::afterTouch

3.2.2.2 int eventPlacement::channelPressure

3.2.2.3 int eventPlacement::controlChange

3.2.2.4 int eventPlacement::noteOff

3.2.2.5 int eventPlacement::noteOn

3.2.2.6 int eventPlacement::pitchWheel

3.2.2.7 int eventPlacement::programChange

The documentation for this struct was generated from the following file:

- **main.c**

3.3 moodWeighting Struct Reference

Data Fields

- char **name** [25]
- int **mode**
- int **tempo**
- int **toneLength**
- int **pitch**

3.3.1 Detailed Description

A struct containing a single moods name and weighting

Parameters

name	the name of the mood
mode	a value -5 to 5 representing this parameters impact on the mood
tempo	a value -5 to 5 representing this parameters impact on the mood
toneLength	a value -5 to 5 representing this parameters impact on the mood

pitch	a value -5 to 5 representing this parameters impact on the mood

3.3.2 Field Documentation

3.3.2.1 int moodWeighting::mode

3.3.2.2 char moodWeighting::name[25]

3.3.2.3 int moodWeighting::pitch

3.3.2.4 int moodWeighting::tempo

3.3.2.5 int moodWeighting::toneLength

The documentation for this struct was generated from the following file:

- **main.c**

3.4 note Struct Reference

Data Fields

- int **tone**
- int **octave**
- int **length**
- int **average**
- int **ticks**

3.4.1 Detailed Description

A struct cotaining data about a single note

Parameters

tone	the tone stored as an integer (C = 0)
octave	which octave, on a piano, the note is in (1 is the deepest, C4 is middle C)
length	the notes length in standard musical notation
average	used in calculating the average tone
ticks	the notes length in ticks

3.4.2 Field Documentation

3.4.2.1 int note::average

3.4.2.2 int note::length

3.4.2.3 int note::octave

3.4.2.4 int note::ticks

3.4.2.5 int note::tone

The documentation for this struct was generated from the following file:

- **main.c**

4 File Documentation

4.1 main.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <dirent.h>
```

Data Structures

- struct **note**
- struct **data**
- struct **moodWeighting**
- struct **eventPlacement**

Macros

- #define **CHARS** 1000
- #define **SCALESIZE** 7

Typedefs

- typedef enum **mode mode**
- typedef enum **tone tone**
- typedef enum **mood mood**

Enumerations

- enum **mode** { **major**, **minor** }
- enum **tone** {
  C, **Csharp**, **D**, **Dsharp**,
  E, **F**, **Fsharp**, **G**,
  Gsharp, **A**, **Asharp**, **B** }
- enum **mood** { **glad**, **sad** }

Functions

- void **checkDirectory** (char ∗, DIR ∗)
- void **printNote** (**note**)
- int **getHex** (FILE ∗, int[ ])
- void **fillSongData** (**data** ∗, int[ ], int)
- int **countPotentialNotes** (int[ ], int)
- void **fillNote** (int, **note** ∗)
- void **settingPoints** (int ∗, int ∗, int ∗, int ∗, **data**, int, **note**[ ], int ∗)
- void **insertMoods** (**moodWeighting**[ ], FILE ∗)
- void **weightingMatrix** (**moodWeighting**[ ], int, int, int, int, int ∗)
- void **findEvents** (int, int[ ], **eventPlacement**[ ], **note**[ ], int ∗, int ∗)
- void **insertPlacement1** (int[ ], int ∗, int, **note**[ ], int ∗, int[ ])
- void **insertPlacement2** (int[ ], int ∗, int)
- int **checkNextEvent** (int[ ], int)

- void **findTicks** (int, int[ ], **eventPlacement**[ ], **note**[ ], int, int ∗, int[ ])
- void **countTicks1** (int[ ], int ∗, int, **note**[ ], int ∗)
- void **countTicks2** (int[ ], int ∗, int, **note**[ ], int ∗)
- void **deltaTimeToNoteLength** (int, int, **note** ∗)
- int **isInScale** (int, int[ ], int)
- int **isInMinor** (int)
- int **isInMajor** (int)
- int **sortToner** (const void ∗, const void ∗)
- void **findMode** (**note** ∗, int, **data** ∗)
- int **FindMoodAmount** (FILE ∗)
- void **printResults** (int, int, int, int, **moodWeighting**[ ], int[ ])
- int **main** (int argc, const char ∗argv[ ])
- int **sortTones** (const void ∗a, const void ∗b)
- void **checkScale** (int scales[ ], int **tone**, int key)
- void **findMode** (**note** noteAr[ ], int totalNotes, **data** ∗**data**)

Variables

- int **AMOUNT_OF_MOODS**

4.1.1 Macro Definition Documentation

4.1.1.1 #define CHARS 1000

4.1.1.2 #define SCALESIZE 7

4.1.2 Typedef Documentation

4.1.2.1 typedef enum mode mode

4.1.2.2 typedef enum mood mood

4.1.2.3 typedef enum tone tone

4.1.3 Enumeration Type Documentation

4.1.3.1 enum mode

Enumerator

major

minor

```
00026 {major, minor} mode;
```

4.1.3.2 enum mood

Enumerator

glad

sad

```
00028 {glad, sad} mood;
```

4.1.3.3 enum tone

Enumerator

C

Csharp

D

Dsharp

E

F

Fsharp

G

Gsharp

A

Asharp

B

```
{C, Csharp, D, Dsharp, E, F, Fsharp, G, Gsharp, A, Asharp, B} tone;
```

4.1.4 Function Documentation

4.1.4.1 void checkDirectory (char ∗ *MIDIfile,* DIR ∗ *dir*)

A function to read music directory and prompt user to choose file

Parameters

MIDIfile	a pointer to a string containing the name of the chosen input file
dir	a pointer to a directory

```
                                            {
  struct dirent *musicDir;
  int musicNumber = -2;

  if((dir = opendir ("./Music")) != NULL) {
    printf(" Mulige numre\n");
    while ((musicDir = readdir (dir)) != NULL){
      if(musicNumber > -1 && musicNumber < 10)
        printf (" %d.  %s\n", musicNumber++, musicDir->d_name);
      else if(musicNumber > -1)
        printf (" %d. %s\n", musicNumber++, musicDir->d_name);
      else
        musicNumber++;
    }
  }
  else{
    perror ("Failure while opening directory");
    exit (EXIT_FAILURE);
  }

  closedir(dir);

  if((dir = opendir ("./Music")) != NULL) {
    printf("\n Indtast det valgte nummer\n ");
    scanf(" %d", &musicNumber);

    for(int i = -2; i <= musicNumber; i++)
      if((musicDir = readdir (dir)) != NULL && i == (musicNumber))
        strcpy(MIDIfile, musicDir->d_name);

    printf("\n Du valgte \n %s\n Hvilket giver disse resultater\n", MIDIfile);
  }
  else{
    perror ("Failure while opening directory");
    exit (EXIT_FAILURE);
  }

  chdir("./Music");
}
```

4.1.4.2 int checkNextEvent (int *hex[ ]*, int *j*)

```
                                      {
  switch (hex[j]){
    case 0x90:
    case 0x80:
    case 0xA0:
    case 0xB0:
    case 0xC0:
    case 0xD0:
    case 0xE0: return 1; break;
    default  : return 0; break;
  }
}
```

4.1.4.3 void checkScale (int *scales[ ]*, int *tone*, int *key*)

Checks if the tone given is within the scale of the key given.

Parameters

scales	An array containing the scalas
tone	An integer representing the tone to be checked
key	Integer representing the key the note is compared to

```
                                                  {
  if(tone < key)
    tone += 12;

  scales[key] = isInMajor(tone - key);
}
```

4.1.4.4 int countPotentialNotes (int *hex[ ]*, int *amount*)

A function to count the number of notes in the entire song

Parameters

hex[]	an array with the stored information from the file
amount	an integer holding the total number of characters in the array

```
                                                {
  int i = 0, res = 0;

  for(i = 0; i < amount; i++){
    if(hex[i] == 0x90){
      res++;
    }
  }

  return res;
}
```

4.1.4.5 void countTicks1 (int *hex[ ]*, int * *i*, int *deltaCounter*, note *noteAr[ ]*, int * *tickCounter*)

Processes events with two parameters, extracting deltatime (and advancing the file pointer)

```
                                                                              {
  noteAr[*tickCounter].ticks = 0;
  int tick = 0;

  while(deltaCounter < 7 && hex[(*i + deltaCounter)] > 0x80)
    tick += ((hex[(*i + deltaCounter++)] - 0x80) << 7);

  tick += hex[(*i + deltaCounter)];
  noteAr[*tickCounter].ticks += tick;
  *i += deltaCounter;
}
```

4.1.4.6 void countTicks2 (int *hex[ ]*, int ∗ *i*, int *deltaCounter*, note *noteAr[ ]*, int ∗ *tickCounter*)

Processes events with one parameter, extracting deltatime (and advancing the file pointer)

```
{
  noteAr[*tickCounter].ticks = 0;
  int tick = 0;

  while(deltaCounter < 6 && hex[(*i + deltaCounter)] > 0x80)
    tick += ((hex[(*i + deltaCounter++)] - 0x80) << 7);

  tick += hex[(*i + deltaCounter)];
  noteAr[*tickCounter].ticks += tick;
  *i += deltaCounter;
}
```

4.1.4.7 void deltaTimeToNoteLength (int *ppqn*, int *size*, note ∗ *noteAr*)

Finds the note length, converted from deltatime to standard musical notation

```
{
  for (int i = 0; i < size; i++){
    double noteLength = ((double) (noteAr[i].ticks)) / ((double) (ppqn/8));

    if (noteLength < 1.5 && noteLength >= 0)
      noteLength = 1;
    else if (noteLength < 3 && noteLength >= 1.5)
      noteLength = 2;
    else if (noteLength < 6 && noteLength >= 3)
      noteLength = 4;
    else if (noteLength < 12 && noteLength >= 6)
      noteLength = 8;
    else if (noteLength < 24 && noteLength >= 12)
      noteLength = 16;
    else
      noteLength = 32;

    noteAr[i].length = noteLength;
  }
}
```

4.1.4.8 void fillNote (int *inputTone*, note ∗ *note*)

A function to fill out each of the structures of type note

Parameters

inputTone	the value of the hexadecimal collected on the "tone"-spot
note∗	a pointer to a note-structure

```
{
  note->tone = inputTone % 12;
  note->average = inputTone;
  note->octave = inputTone / 12;
}
```

4.1.4.9 void fillSongData (data ∗ *data*, int *hex[ ]*, int *numbersInText*)

! A function, that fills out the song data

Parameters

∗*data*	a pointer to a structure containing the tempo and mode of the song
hex[]	the array of integers read from the file
numbersInText	the total amount of integers in the array

```
{
  data->ppqn = (hex[12] << 8) + hex[13];

  /*Find the mode of the song, initialised as minor atm*/
  for(int j = 0; j < numbersInText; j++)
```

```
    /* finds the tempo */
    if(hex[j] == 0xff && hex[j+1] == 0x51 && hex[j+2] == 0x03)
      data->tempo =  60000000/((hex[j+3] << 16) | (hex[j+4] << 8) | (hex[j+5]));
}
```

4.1.4.10 void findEvents (int *numbersInText,* int *hex[ ],* eventPlacement *placement[ ],* note *noteAr[ ],* int ∗ *size,* int ∗ *amountOfNotes*)

Searches the file for events and stores their placement in an array of **eventPlacement** (p. 2) structs

```

            {
  int noteOff = 0, noteOn = 0, afterTouch = 0, controlChange = 0,
      programChange = 0, channelPressure = 0, pitchWheel = 0, notes[numbersInText];

  for(int j = 0; j < numbersInText; j++)
    switch (hex[j]){
      case 0x90: insertPlacement1(hex, &placement[noteOn++].noteOn, j, noteAr, amountOfNotes, notes);
         break;
      case 0x80: insertPlacement1(hex, &placement[noteOff++].noteOff, j, noteAr, amountOfNotes, notes);
         break;
      case 0xA0: insertPlacement1(hex, &placement[afterTouch++].afterTouch, j, noteAr, amountOfNotes, notes
);       break;
      case 0xB0: insertPlacement1(hex, &placement[controlChange++].controlChange, j, noteAr, amountOfNotes,
 notes); break;
      case 0xC0: insertPlacement2(hex, &placement[programChange++].programChange, j);
         break;
      case 0xD0: insertPlacement2(hex, &placement[channelPressure++].channelPressure, j);
         break;
      case 0xE0: insertPlacement1(hex, &placement[pitchWheel++].pitchWheel, j, noteAr, amountOfNotes, notes
);       break;
      default  :
         break;
    }
  findTicks(numbersInText, hex, placement, noteAr, noteOn, size, notes);
}
```

4.1.4.11 void findMode (note ∗ , int , data ∗)

4.1.4.12 void findMode (note *noteAr[ ],* int *totalNotes,* data ∗ *data*)

A function to find the mode of the song by first calculating the tone span over sets of notes in the song, and then comparing it to the definition of minor and major keys.

Parameters

noteAr	An array of all the notes in the entire song
totalNotes	The number of notes in the song
data	The song data

```
                                                        {
  int majors[12] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, minors[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  int x = 0, y = 0, z = 0, bar[4], sizeBar = 4, tempSpan = 999, span = 999, keynote = 0,
mode = 0, tempNote = 0;

  for(x = 0; x < totalNotes; x++){
    tempNote = noteAr[x].tone;

    for(y = C; y <= B; y++)
      if(majors[y])
        checkScale(majors, tempNote, y);
  }

  for(y = 0; y < 12; y++){
    z = y;

    if(majors[z]){
      if((z - 3) < 0)
        z += 12;

      minors[z-3] = 1;
    }
  }

  z = 0;  x = 0;

  /*Goes through all notes of the song and puts them into an array, 4 at a time*/
  while(x < totalNotes){
    z = x;

    for(y = 0; y < sizeBar; y++, z++){
      if(z < totalNotes)
        bar[y] = noteAr[z].tone;
      else
        sizeBar = y;
    }

    if(y == sizeBar){
      span = 999;
      /*Sort notes in ascending order*/
      qsort(bar, sizeBar, sizeof(tone), sortTones);

      /*Finds the lowest possible tonespan over the array of 4 notes*/
      for(z = 0; z < sizeBar; z++){
          if((z + 1) > 3)
          tempSpan = (bar[(z+1)%4]+12)-bar[z] + bar[(z+2)%4]-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
        else if((z + 2) > 3)
          tempSpan = bar[(z+1)]-bar[z] + (bar[(z+2)%4]+12)-bar[(z+1)%4] + bar[(z+3)%4]-bar[(z+2)%4];
          else if((z +3) > 3)
          tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + (bar[(z+3)%4]+12)-bar[z];
          else
          tempSpan = bar[(z+1)]-bar[z] + bar[(z+2)]-bar[(z+1)] + bar[(z+3)]-bar[(z+2)];
          if(tempSpan < span && (majors[bar[z]] || minors[bar[z]])){
          span = tempSpan;
          keynote = bar[z];
        }
      }

      mode += isInScale(keynote, bar, sizeBar);
      x++;
    }
  }

  /*outputs result directly to the data struct*/
  if(mode > 0)
    data->mode = major;
  else if(mode < 0)
    data->mode = minor;
}
```

4.1.4.13 int FindMoodAmount (FILE * *moods*)

```
                               {
  int i = 1;

  while(fgetc(moods) != EOF)
    if(fgetc(moods) == '\n')
      i++;

  rewind(moods);
  return i;
}
```

4.1.4.14 void findTicks (int *numbersInText,* int *hex[ ],* eventPlacement *placement[ ],* note *noteAr[ ],* int *noteOn,* int * *size,* int *notes[ ]*)

```

                {
  int tickCounter = 0, deltaCounter1 = 3, deltaCounter2 = 2;

  for(int j = 0; j < noteOn; j++){
    for(int i = placement[j].noteOn; i < numbersInText; i++){
      if(hex[i] == 0x80){
        if(hex[i + 1] == notes[j]){
          tickCounter++;
          break;
        }
        else
          countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
      }
      else if(hex[i] == 0xA0){
        if(hex[i + 1] == notes[j] && hex[i + 2] == 0x00){
          tickCounter++;
          break;
        }
        else
          countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
      }
      else if(hex[i] == 0xD0){
        if(hex[i + 1] == 0x00){
          tickCounter++;
          break;
        }
        else
          countTicks2(hex, &i, deltaCounter2, noteAr, &tickCounter);
      }
      else if(hex[i] == 0xC0)
        countTicks2(hex, &i, deltaCounter2, noteAr, &tickCounter);
      else
        countTicks1(hex, &i, deltaCounter1, noteAr, &tickCounter);
    }
  }

  *size = tickCounter;
}
```

4.1.4.15 int getHex (FILE ∗ *f*, int *hexAr[]*)

A function, that retrieves the hexadecimals from the files and also returns the number of files

Parameters

∗*f*	a pointer to the file the program is reading from
hexAr[]	an array of integers, that the information is stored in

```
                                {
  int i = 0, c;

  while( (c = fgetc(f)) != EOF && i < CHARS){
    hexAr[i] = c;
    i++;
  }

  return i;
}
```

4.1.4.16 void insertMoods (moodWeighting *moodArray[]*, FILE ∗ *moods*)

Inserts the weighting of each mood in an array of structs, as read from a designated file.

Parameters

moodArray	The array moods are stored in
moods	the file to be read

```
                                                        {
  for(int i = 0; i < AMOUNT_OF_MOODS; i++)
    fscanf(moods, "%s %d %d %d %d", moodArray[i].name , &moodArray[i].mode,
                                    &moodArray[i].tempo, &moodArray[i].toneLength,
                                    &moodArray[i].pitch);
}
```

4.1.4.17 void insertPlacement1 (int *hex[]*, int ∗ *place*, int *j*, note *noteAr[]*, int ∗ *amountOfNotes*, int *notes[]*)

Starts in the hex which are investigated and looks forward to find a perspective. It goes to an assumed deltatime and finds the length of it. Thereafter it checks the next hex after the deltatime to make sure it is an event. If that is the case it stores the hex which is investegated in the first place. Furthermore if it is a noteOn event it stores the hex which is the note, processes the note and counts amount of notes.

```
                                                                                                    {
  int i = 3;

  while(i < 7 && hex[(j + i++)] > 0x80);

  if(checkNextEvent(hex, (j + i))){
    *place = j;
    if(hex[j] == 0x90){
      notes[*amountOfNotes] = hex[j + 1];
      fillNote(hex[j + 1], &noteAr[*amountOfNotes]);
      *amountOfNotes += 1;
    }
  }
}
```

4.1.4.18 void insertPlacement2 (int *hex[ ],* int ∗ *place,* int *j*)

Does the same as insertPlacement1, but for events with 1 parameter.

```
                                                   {
  int i = 2;

  while(i < 6 && hex[(j + i++)] > 0x80);

  if(checkNextEvent(hex, (j + i)))
    *place = j;
}
```

4.1.4.19 int isInMajor (int *toneLeap*)

A function to check if the given tone leap is in the major scale.

Parameters

toneLeap	An integer describing the processed tone leap

Returns

a boolean value, returns 1 if the tone leap is in the major scale, 0 if it's not.

```
                           {
  int major[] = {0, 2, 4, 5, 7, 9, 11};

  for(int i = 0; i < SCALESIZE; i++)
    if(toneLeap == major[i])
      return 1;

  return 0;
}
```

4.1.4.20 int isInMinor (int *toneLeap*)

A function to check if the given tone leap is in the minor scale.

Parameters

toneLeap	An integer describing the processed tone leap

Returns

a boolean value, returns 1 if the tone leap is in the minor scale, 0 if it's not.

```
                           {
  int minor[] = {0, 2, 3, 5, 7, 8, 10};

  for(int i = 0; i < SCALESIZE; i++)
    if(toneLeap == minor[i])
      return 1;

  return 0;
}
```

4.1.4.21 int isInScale (int *keytone,* int *otherTones[ ],* int *size*)

A function to check if a given scale in given keytone corresponds with the tones in the rest of the song.

Parameters

keytone	The keytone of the processed scale
otherTones	An array of the rest of the tones, which the function compares to the keytone and mode
size	The number of tones in the otherTones array

Returns

a boolean value, returns 1 if the mode is major, -1 if it's minor and 0, if wasn't possible to decide.

```
                           {
  int toneLeap, isMinor = 1, isMajor = 1;

  for(int i = 0; i < size; i++){
    if(otherTones[i] < keytone)
      otherTones[i] += 12;

    toneLeap = otherTones[i] - keytone;

    if(isMinor)
      isMinor = isInMinor(toneLeap);

    if(isMajor)
      isMajor = isInMajor(toneLeap);
  }

  if(isMinor && isMajor)
    return 0;
  else if(isMinor)
    return -1;
  else if(isMajor)
    return 1;

  return 0;
}
```

4.1.4.22 int main (int *argc,* const char ∗ *argv[]*)

```
                   {
  DIR *dir = 0;
  FILE *f;
  char MIDIfile[25];
  /*Variables*/
  int numbersInText = 0, notes, size = 0, mode = 5, tempo = 5, toneLength = 5, pitch = 5, amountOfNotes = 0
;
  FILE* moods = fopen("moods.txt", "r");

  if(moods == NULL){
    perror("Error: moods missing ");
    exit(EXIT_FAILURE);
  }

  AMOUNT_OF_MOODS = FindMoodAmount(moods);
  moodWeighting moodArray[AMOUNT_OF_MOODS];
  data data = {0, major, D};

  if (argv[1] == NULL){
    checkDirectory(MIDIfile, dir);
    f = fopen(MIDIfile, "r");

    if(f == NULL){
      perror("Error opening file");
      exit(EXIT_FAILURE);
    }
  }
  else if(argv[1] != NULL){
    f = fopen(argv[1],"r");

    if(f == NULL){
      perror("Error opening file");
      exit(EXIT_FAILURE);
    }
  }

  closedir (dir);
  int *hex = (int *) malloc(CHARS * sizeof(int));

  if(hex == NULL){
    printf("Memory allocation failed, bye!");
    exit(EXIT_FAILURE);
  }

  /*Reading the data from the file*/
  numbersInText = getHex(f, hex);
  fillSongData(&data, hex, numbersInText);
  notes = countPotentialNotes(hex, numbersInText);
  note *noteAr = (note*) malloc(notes * sizeof(note));

  if(noteAr == NULL){
    printf("Memory allocation failed, bye!");
    exit(EXIT_FAILURE);
  }

  eventPlacement placement[numbersInText];
  int result[AMOUNT_OF_MOODS];
  findEvents(numbersInText, hex, placement, noteAr, &size, &amountOfNotes);
  deltaTimeToNoteLength(data.ppqn, size, noteAr);
  insertMoods(moodArray, moods);
  findMode(noteAr, amountOfNotes, &data);
  settingPoints(&mode, &tempo, &toneLength, &pitch, data, amountOfNotes, noteAr, &size);
  weightingMatrix(moodArray, mode, tempo, toneLength, pitch, result);

  /*Clean up and close*/
  fclose(f);
  free(hex);
  free(noteAr);

  /* Print results */
  printResults(mode, tempo, toneLength, pitch, moodArray, result);

  return 0;
}
```

4.1.4.23 void printNote (note *note*)

A function to print the note

Parameters

note	the note structure to be printed

```
                          {
  printf("Tone: ");

  switch (note.tone){
    case C      : printf("C") ; break;
    case Csharp: printf("C#"); break;
    case D      : printf("D") ; break;
    case Dsharp: printf("D#"); break;
    case E      : printf("E") ; break;
    case F      : printf("F") ; break;
    case Fsharp: printf("F#"); break;
    case G      : printf("G") ; break;
    case Gsharp: printf("G#"); break;
    case A      : printf("A") ; break;
    case Asharp: printf("A#"); break;
    case B      : printf("B") ; break;
    default     : printf("Undefined note"); break;
  }

  printf(", octave: %d\n", note.octave);
}
```

4.1.4.24 void printResults (int *mode,* int *tempo,* int *toneLength,* int *pitch,* moodWeighting *moodArray[],* int *result[]*)

Prints relevant information about the song. Finds and prints the mood with the highest score, and in the case of using the default sad/happy scale, scales the values to fit on the 51 point sliding scale

```
                                                                                                              {
  printf("\n\n\n");
  printf(" Mode:");
  printf("%10d\n", mode);
  printf(" Tempo:");
  printf("%9d\n", tempo);
  printf(" Tone length:");
  printf("%3d\n", toneLength);
  printf(" Pitch:");
  printf("%9d\n", pitch);
  printf("\n\n\n                                        WEIGHTINGS\n");
  printf("                            Mode | Tempo | Tone length | Pitch\n");

  for(int i = 0; i < AMOUNT_OF_MOODS; i++){
    printf(" %s", moodArray[i].name);
    for(int j = strlen(moodArray[i].name); j < 26; j++)
      printf(" ");
    if(moodArray[i].mode > -1)
      printf(" ");
    printf(" %d", moodArray[i].mode);
    for(int j = 0; j < 2; j++)
      printf(" ");
    printf("| ");
    if(moodArray[i].tempo > -1)
      printf(" ");
    printf(" %d", moodArray[i].tempo);
    for(int j = 0; j < 3; j++)
      printf(" ");
    printf("|    ");
    if(moodArray[i].toneLength > -1)
      printf(" ");
    printf(" %d", moodArray[i].toneLength);
    for(int j = 0; j < 6; j++)
      printf(" ");
    printf("|  ");
    if(moodArray[i].pitch > -1)
      printf(" ");
    printf(" %d\n", moodArray[i].pitch);
  }

  printf("\n\n\n");

  for(int i = 0; i < AMOUNT_OF_MOODS; i++){
    if(mode < 0)
      printf(" %d * ", mode);
    else
      printf(" %d * ", mode);
    if(moodArray[i].mode < 0)
      printf("%d + ", moodArray[i].mode);
    else
      printf(" %d + ", moodArray[i].mode);
    if(tempo < 0)
      printf("%d * ", tempo);
    else
      printf(" %d * ", tempo);
    if(moodArray[i].tempo < 0)
      printf("%d + ", moodArray[i].tempo);
    else
      printf(" %d + ", moodArray[i].tempo);
    if(toneLength < 0)
      printf("%d * ", toneLength);
    else
      printf(" %d * ", toneLength);
    if(moodArray[i].toneLength < 0)
      printf("%d + ", moodArray[i].toneLength);
    else
      printf(" %d + ", moodArray[i].toneLength);
    if(pitch < 0)
      printf("%d * ", pitch);
    else
      printf(" %d * ", pitch);
    if(moodArray[i].pitch < 0)
      printf("%d = ", moodArray[i].pitch);
    else
      printf(" %d = ", moodArray[i].pitch);
    if(result[i] < 0)
      printf("%d\n", result[i]);
    else
      printf(" %d\n", result[i]);
  }

  int moodOfMelodi = 0, test = 0;

  for(int i = 0; i < AMOUNT_OF_MOODS; i++)
    if(moodOfMelodi < result[i])
      moodOfMelodi = i;

  if(!strcmp(moodArray[moodOfMelodi].name, "Happy")){
    printf("\n\n\n Sad ");

    while(test < 51){
      if(test == 25)
        printf("|");
      else if(test == ((result[moodOfMelodi] / 2) + 26))
        printf("[]");
      else
        printf("-");

      test++;
    }

    printf(" Happy\n\n\n");
  }
  else if(!strcmp(moodArray[moodOfMelodi].name, "Sad")){
    printf("\n\n\n Sad ");

    while(test < 51){
      if(test == 25)
        printf("|");
      else if(test == ((int)(-((result[moodOfMelodi]) / 2.4)) + 26))
        printf("[]");
      else
        printf("-");

      test++;
    }

    printf(" Happy\n\n\n");
  }

  printf("\n The mood of the melody is %s\n", moodArray[moodOfMelodi].name);
}
```

4.1.4.25 void settingPoints (int ∗ *mode,* int ∗ *tempo,* int ∗ *length,* int ∗ *octave,* data *data,* int *notes,* note *noteAr[ ],* int ∗ *size*)

A function to insert points into integers based on the data pulled from the file

Parameters

mode,along	with tempo, length and octave contains the points
data	contains the song data
notes	contains the amount of notes in the song
note	contains an array of the specific notes

```

        {
  int deltaTime = 0, combined = 0, averageNote = 0;

  switch(data.mode){
    case minor: *mode = -5; break;
    case major: *mode = 5; break;
    default: *mode = 0; break;
  }

  if(data.tempo < 60)
    *tempo = -5;
  else if(data.tempo >= 60 && data.tempo < 70)
    *tempo = -4;
  else if(data.tempo >= 70 && data.tempo < 80)
    *tempo = -3;
  else if(data.tempo >= 80 && data.tempo < 90)
    *tempo = -2;
  else if(data.tempo >= 90 && data.tempo < 100)
    *tempo = -1;
  else if(data.tempo >= 100 && data.tempo < 120)
    *tempo =  0;
  else if(data.tempo >= 120 && data.tempo < 130)
    *tempo =  1;
  else if(data.tempo >= 130 && data.tempo < 140)
    *tempo =  2;
  else if(data.tempo >= 140 && data.tempo < 150)
    *tempo =  3;
  else if(data.tempo >= 150 && data.tempo < 160)
    *tempo =  4;
  else if(data.tempo >=  160)
    *tempo =  5;

  for(int i = 0; i < notes; i++)
    combined += noteAr[i].length;

  deltaTime = combined/notes;

  if (deltaTime < 1.5 && deltaTime >= 0)
    *length = 5;
  else if (deltaTime < 3 && deltaTime >= 1.5)
    *length = 4;
  else if (deltaTime < 5 && deltaTime >= 4)
    *length = 3;
  else if (deltaTime < 6 && deltaTime >= 5)
    *length = 2;
  else if (deltaTime < 9 && deltaTime >= 6)
    *length = 1;
  else if (deltaTime < 12 && deltaTime >= 9)
    *length = 0;
  else if (deltaTime < 16 && deltaTime >= 12)
    *length = -1;
  else if (deltaTime < 20 && deltaTime >= 16)
    *length = -2;
  else if (deltaTime < 24 && deltaTime >= 20)
    *length = -3;
  else if (deltaTime < 28 && deltaTime >= 24)
    *length = -4;
  else
    *length = -5;

  combined = 0;

  for (int i = 0; i < notes; i++)
    combined += noteAr[i].average;

  averageNote = combined/notes;

  if(averageNote <= 16)
    *octave = -5;
  else if(averageNote >= 17 && averageNote <= 23)
    *octave = -4;
  else if(averageNote >= 24 && averageNote <= 30)
    *octave = -3;
  else if(averageNote >= 31 && averageNote <= 37)
    *octave = -2;
  else if(averageNote >= 38 && averageNote <= 44)
    *octave = -1;
  else if(averageNote >= 45 && averageNote <= 51)
    *octave = 0;
  else if(averageNote >= 52 && averageNote <= 58)
    *octave = 1;
  else if(averageNote >= 59 && averageNote <= 65)
    *octave = 2;
  else if(averageNote >= 66 && averageNote <= 72)
    *octave = 3;
  else if(averageNote >= 73 && averageNote <= 79)
    *octave = 4;
  else if(averageNote >=80)
    *octave = 5;
}
```

4.1.4.26 int sortToner (const void ∗ , const void ∗)

4.1.4.27 int sortTones (const void ∗ *a*, const void ∗ *b*)

A function to sort integers in ascending order, used by qsort

```
                                          {
  return (*(int *)a - *(int *)b);
}
```

4.1.4.28 void weightingMatrix (moodWeighting *moodArray[ ]*, int *mode*, int *tempo*, int *toneLength*, int *pitch*, int ∗ *result*)

Vector matrix multiplication. Receives an array of moods, the various parameters of the song and a pointer to an array where the results will be stored. The song data is multiplied onto each moods weighting and then stored.

Parameters

moodArray	an array containing the weighting for all moods
result	an array for holding the songs scores as per each mood
mode	along with temp, toneLength and pitch, this variable contains a score -5 to 5 for how that facet of the song is.

```

{
  for(int i = 0; i < AMOUNT_OF_MOODS; i++)
    result[i] = 0;

  for(int i = 0; i < AMOUNT_OF_MOODS; i++){
    result[i] += (moodArray[i].mode * mode);
    result[i] += (moodArray[i].tempo * tempo);
    result[i] += (moodArray[i].toneLength * toneLength);
    result[i] += (moodArray[i].pitch * pitch);
  }
}
```

4.1.5 Variable Documentation

4.1.5.1 int AMOUNT_OF_MOODS

Index