MPL

Music Processing Language

Tutorial

Project Manager:	Bo Wang (bw2450)
Language Guru:	Yilin Xiong (yx2274)
System Architect:	Shengyi Lin (sl3759)
System Integrator:	Ying Tan (yt2443)
Verification and Validation:	Mengting Wu (mw2987)

1. Getting Started

First, we will show how to write a program to print the words: hello, world to show the process of creating the program text, compiling it, loading it, running it and finding out where the output went. With these mechanical details mastered, everything else is comparatively easy. In MPL, the program to print “hello world” is

Example:

```
void main() {
     print (“Hello World!\n”);
}
```

Just how to run this program depends on the system using. As a specific example on the UNIX operating system, you must create the program in a file whose name ends in “.mpl”, such as hello.mpl, then compile it with the command

```
mpl hello.mpl hello
```

If you haven’t botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called hello. If you run hello by typing the command

```
hello
```

It will print : **Hello World!**

On other systems, the rules will be different; check with a local expert.

Now, for some explanations about the program itself, a MPL program, whatever its size, consists of functions and variables. A function contains statements that specify the computing operations to be done, and variables store values used during the computation. Our functions are mainly C like, and our example is a function named main. Normally you are at liberty to give functions whatever names you like, but “main" is special - your program begins executing at the beginning of main. This means that every program must have a main somewhere and it will usually call other functions to help perform its job.

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function print with the argument “Hello World!\n”. The print() is a function that prints output, in this case the string of characters between the double quotes.

A sequence of characters in double quotes, such as “Hello world!\n”, is a string constant or string literal in MPL. And the sequence \n in the string is the C-like notation for the newline character, when printed advances the output to the left margin is on next line.

2. Note Declarations and Initialization

The basic component of music is its notes. To create a new note, we can use the declaration statement as follows:

Example:

```
Note note1 = Note(); // Create a new note named note1
```

The basic type Note has four attributes pitch, duration, starttime, strength. To set the default value of the new created note, we can use the statement as follows：

Example:

```
setNoteDefault(C4, 250, 0, 200);
// set pitch default value as C4, set duration value as
// 250, set startime value as 0, set strength value as 200
```

After setting the default values manually, once we create a new note again by the declaration

Example:

```
Note note2 = Note(); // Create a new note named note2
```

Then the attribute values of note2 will be the manually-set default values instead of the systematically-set default values.

To initialize Note, we can use the initialization statements as follows:

Example:

```
Note(G4);
// create a note with the pitch value as G4, and the values
// of its duration, starttime, strength as default

Note(G4,500);
// create a note with the pitch value as G4, the duration
// value as 500, and the values of its starttime,strength
// as default

Note(G4,500,10,300);
// create a note with the pitch value as G4, the duration
// value as 500, the starttime value as 10, and the
// strength value as 300
```

3. Melody Declarations and Initialization

Melody in MPL is defined as a sequential list of notes, which can be manipulated as a whole. To create a new melody, we can use the declaration statement as follows.

Example:

```
Melody melody0 = Melody() // construct a new melody object
```

To initialize a melody, we can use the initialization statements as follows:

Example:

```
Melody(notes)
// construct a new Melody object  composed of the input the
// note array notes[]

Melody(melody0)
// construct a new Melody object which has same attributes
// as the input melody0.
```

4. Melody Modification

4.1. Melody Addition:

User can use the operation: + to overlay the new music on the original one. In this case, we assume that the following forms of original music and new music:

Original music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Violin: 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - ||
```

New music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - | 5 5 4 4 |
       3 3 2 - | 5 5 4 4 | 3 3 2 - ||
Violin: 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - | 5 - 4 - |
       3 - 2 - | 5 - 4 - | 3 - 2 - ||
```

The example to combine these music are shown below:

```
void main(){

    String path = "~/Desktop/twinkle_twinkle.midi";
    Music music = read(path);
    // read in the midi file from path

    Track tracks[] = music.getTracks(); //get tracks array

    /* 5 5 4 4 | 3 3 2 - | 5 5 4 4 | 3 3 2 - ||*/

    setNoteDefault(G4, 250, 0, 200);
    // set pitch default value as G4, set duration
    // value as 250, set startime value as 0, set
    // strength value as 200

```

```
     Note notes0[] = {Note(), Note(), Note(F4),  Note(F4),
        Note(E4),  Note(E4), Note(D4, 500), Note(), Note(),
        Note(F4),  Note(F4),  Note(E4),  Note(E4),
        Note(D4, 500)};
     // initialize notes array
     /* 5 - 4 - | 3 - 2 - | 5 - 4 - | 3 - 2 - ||*/
     setNoteDefault(G4, 250, 0, 200);
     // set pitch default value as G4, set duration value
     // as 250, set startime value as 0, set strength value
     // as 200

     Note notes1[] = {Note(), Note(F4), Note(E4),  Note(D4),
        Note(), Note(F4), Note(E4), Note(D4)};
     // initialize notes array

     Melody melody0(notes0);// Initialize melody0
     Melody melody1(notes1);//Initialize melody1

     tracks[0].getMelody() += melody0;
     // concatenate melody0 to tracks[0]'s original melody
     tracks[1].getMelody() += melody1;
     // concatenate melody1 to tracks[1]'s original melody

     write(music, "~/Desktop/new_twinkle_twinkle.midi");
     // write the new music to output path
}
```

4.2 Melody multiplication

User can use the operation * to superimpose melodies to a new melody. In this case, we assume that the following forms of original music and new music:

Original music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - | 5 5 4 4 |
Violin: 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - | 5 - 4 - |

Piano:  3 3 2 - | 5 5 4 4 | 3 3 2 - ||
Violin: 3 - 2 - | 5 - 4 - | 3 -  2 -||
```

New music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - | 5 5 4 4 |
Violin: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - | 5 5 4 4 |
        1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - | 5 - 4 - |

Piano:  3 3 2 - | 5 5 4 4 | 3 3 2 - ||
Violin: 3 3 2 - | 5 5 4 4 | 3 3 2 - ||
        3 - 2 - | 5 - 4 - | 3 - 2 - ||
```

The example to create the new music is shown below:

```
void main(){
   string path = "~/Desktop/twinkle_twinkle.midi";
   Music music = read(path); //read in the music midi file

   Track tracks[] = music.getTracks();
   // tracks[0] is for the melody with timbre value
   // of Piano and tracks[1] is of the melody with
   // timbre value of Violin

   // Original melody:
   // 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - |
   // 5 - 4 - | 3 - 2 - | 5 - 4 - | 3 - 2 - ||

   // After multiplication:
   // 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - |
   // 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - |
   //
```

```
    // 5 5 4 4 | 3 3 2 - | 5 5 4 4 | 3 3 2 - ||
    // 5 - 4 - | 3 - 2 - | 5 - 4 - | 3 -  2 -||

    tracks[1].getMelody() *= tracks[0].getMelody();
    // get the melody of the piano track and superimpose
    // it to the melody of the violin track to set as
    // the new melody of the violin track

    write(music, "~/Desktop/new_twinkle_twinkle.midi");
    // export to a midi file
}
```

4.3 Melody Modification

4.3.1 Note Insertion

User can use the function addNote() to insert single note into the existing melody with user-specified attributes. In this case, the note to be inserted is defined as follows:

```
setNoteDefault(C4, 200, 0, 200);
Note new_note(C4);
```

The original music is set by the following statement as the following form:

```
setNoteDefault(C4, 200, 0, 200);
Note notes[] = {Note(C4), Note(D4),
               Note(E4), Note(F4)};
Melody melody(notes);
```

Original Melody:

```
1 2 3 4||
```

The example of inserting a note into the original melody is shown below:

```
melody.addNote(new_note);
```

After insertion, the new melody will be as follows:

`New Melody :`

```
// The melody becomes 1 2 3 4|
//                    1      |
```

4.3.2 Note Deletion

This method is used for deleting the specific Note object at the specified position in the Melody object. The index should be smaller than the note array's length, and larger than 0 inclusively:

The original melody is set by the following statement as the following form:

```
Note notes[] = {Note(C4), Note(D4), Note(E4), Note(F4)};
Melody melody(notes);
```

So we get the following melody: `1 2 3 4||`

The example to delete a specified note from the melody is shown below:

`Example:`

```
// The melody becomes 1 2 0 4|
melody.deleteNote(2);
```

5. Track Declarations and Initialization

Track in MPL is defined as the container of melodies with a specified attribute named timbre.

To create a new track ,we can use the declaration statement as follows:

Example:

```
Track track1 = Track(); //construct a new track object
```

To initialize a track, we can use the initialization statements as follows:

Example:

```
Track(melody0);
// construct a new Track object based on melody0 with the
// default timbre as PIANO

Track(melody0, VIOLIN);
// construct a new Track object based on melody0 and set the
// timbre as VIOLIN

Track(track1);
// construct a new Track object which has same attributes as track1
```

6. Music Declarations and Initialization, Reading and Writing

To create a new music, we can use the declaration statement as follows:

Example:

```
Music music = Music(); // construct a new Music object
```

To initialize a music, we can use the initialization statements as follows:

Example:

```
Music(tracks);
// construct a new Music object based on the tracks[] array

Music(music0);
// clone an existing Music object to a new Music object
```

MPL supports importing external midi files directly for further modification, the function read() can be applied to read files with the statements as follows:

Example:

```
string path = "~/Desktop/twinkle_twinkle.midi";
Music music = read(path); // read in the midi file from path
```

After finishing the music composition, you can export your music with the function write() as a midi file to play in the midi player. The statement is as follows:

Example:

```
write(music, "~/Desktop/new_twinkle_twinkle.midi");
// write the new music to output path
```

7. Music Creation

The example of writing a complete music program is shown below:
In this program, we want to write a piece of music such as:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Violin: 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - ||
```

The program below shows how to write it.

```
void main(){

     // The following two-line statements are used for
     // initialize required objects for the music:
     /* 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - || */

     setNoteDefault(C4, 250, 0, 200);
     // set pitch default value as C4, set duration value as 250,
     // set startime value as 0, set strength value as 200


      Note notes0[] = {Note((), Note(), Note(G4), Note(G4),
          Note(A4), Note(A4),Note(G4, 500), Note(F4), Note(F4),
          Note(E4), Note(E4),Note(D4), Note(D4), Note(C4, 500)};
      // Initialize notes array




      // The following two-line statements are used for initialize
      // required objects for the music:
      /* 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - || */

      setNoteDefault(C4, 500, 0, 200);
      // set pitch default value as C4, set duration value as
      // 250, set starttime value as 0, set strength value as 200


      Note  notes1[]  =  {Note(),  Note(G4),  Note(A4),Note(G4),
      Note(F4), Note(E4),Note(D4), Note()};
      // Initialize notes array for the violin part

      Melody melody0(notes0), melody1(notes1);
      // Initialize melody object

      Track tracks[] = {Track(melody0, PIANO),
         Track(melody1, VIOLIN)};
      // Initialize track[] array, which combines the track
      // from melody0 object with PIANO timbre, and another
      // track from melody1 object with VIOLIN timbre.

      Music music(tracks); // Initialize music object

      write(music, "~/Desktop/twinkle_twinkle.midi");
      // output a midi file.
}
```

We provide the ways for user to create music conveniently. Not only user can write a piece of music, but can also modify music in different approaches, such as: music addition, music multiplication and change timbre of track.

7.1 Music Addition

User can use the arithmetic operator + to concatenate musics to a new music. In this case, we assume that the following forms of original music and new music:

Original music1:

```
Piano: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
```

Original music2:

```
Piano:  5 5 4 4 | 3 3 2 - | 5 5 4 4 | 3 3 2 - ||
Violin: 5 5 4 4 | 3 3 2 - | 5 5 4 4 | 3 3 2 - ||
```

New music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - | 0 - - - |
        0 - - - | 0 - - - | 0 - - - ||
Piano:  0 - - - | 0 - - - | 0 - - - | 0 - - - | 5 5 4 4 |
        3 3 2 - | 5 5 4 4 | 3 3 2 - ||
Violin: 0 - - - | 0 - - - | 0 - - - | 0 - - -| 5 - 4 - |
        3 - 2 - | 5 - 4 - | 3 -  2 -||
```

The example to create the new music is shown below:

```
void main(){
    string path0 = “~/Desktop/twinkle_twinkle0.midi”;
    string path1 = “~/Desktop/twinkle_twinkle1.midi”;
    Music music0 = read(path1); // reading music0
    Music music1 = read(path2); // reading music1

    Music music = music0 + music1;
    // concatenate music0 and music1 to create a new music.
    // music0 is played first then follows music1

    write(music, “~/Desktop/new_twinkle_twinkle.midi”);
    // export the midi file
}
```

7.2 Music multiplication

User can use the operation * to superimpose musics to a new music. In this case, we assume that the following forms of original music and new music:

Original music1:

```
Piano: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
```

Original music2:

```
Violin: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Piano:  1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - ||
```

New music:

```
Piano:  1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Violin: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Piano:  1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - ||
```

The example to create the new music is shown below:

```
void main(){
    string path0 = “~/Desktop/twinkle_twinkle0.midi”;
    string path1 = “~/Desktop/twinkle_twinkle1.midi”;
    Music music0 = read(path1); // reading music0
    Music music1 = read(path2); // reading music1
    Music music = music0 * music1;
    // superimpose music0 and music1 together to create a new music

    write(music, “~/Desktop/new_twinkle_twinkle.midi”);
    // export the midi file
}
```

7.3 Change timbre

User can use the method setTimbre() to change the timbre of the entire melody to create a new music. In this case, we assume that the following forms of original music and new music:

Original music:

```
Piano: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Piano: 1 - 5 - | 6 - 5 - | 4 - 3 - | 2 - 1 - ||
```

Original music:

```
Violin: 1 1 5 5 | 6 6 5 - | 4 4 3 3 | 2 2 1 - ||
Violin: 1 -  5 -  | 6 - 5 -  | 4 - 3 -  | 2 - 1 -  ||
```

The example to create the new music is shown below:

```
void main(){
    Music music = read("~/Desktop/twinkle_twinkle.midi");
    //reading music

    for(int i = 0; i < music.getNumberOfTracks(); i++) {
        music.getTrack(i).setTimbre(VIOLIN);
    }
    // for each track in the music, set its timbre to VIOLIN

    write(music, "~/Desktop/new_twinkle_twinkle.midi");
    // export the midi file
}
```