The Design of a C++ Class Library for MIDI Applications

Greg Sepeesi
chMIDI, P.O. Box 11586, Lynchburg, VA 24506; (804) 385-0477; fax (804) 385-0434

Bill Punch
punch@cps.msu.edu
Computer Science Dept, A714 Wells Hall, Michigan State University, East Lansing, MI 48824

Using the manipulation of ASCII text as an analogy, this report describes the design of a set of C++ classes that manipulate MIDI data.

1. Overview

1.1 C++

C++ [3,4] is a general purpose programming language which is a superset of the C programming language. C supports procedural programming, a design method which results in a collection of potentially reusable procedures. C++ supports procedural programming and also object oriented programming, a design method which results in a collection of potentially reusable abstract data types, called classes. A data type is "abstract" when written so users can focus on the operations allowed on the data, instead of the structure of the data itself. For instance, the push and pop operations abstractly define a stack regardless of the underlying data structure, such as an array or a linked list.

Although a C++ program is no more capable than a C program, C++ provides mechanisms which make it easier to reuse code. This suggests that once these mechanisms are learned, more C++ code will be reused and C++ programming will take less time.

1.2 MIDI

MIDI [1] is a hardware and protocol standard for the transmission and reception of music information. The data is sent serially and asynchronously at 312.5K bits per second over an optically isolated 5 mA current loop. MIDI messages can be grouped into four categories.

A channel message is a 2 or 3 byte message containing a 4-bit field defining the message's destination address, called the channel number. An example is a message to instruments receiving channel 5 to play middle C.

A system common message is a 1, 2, or 3 byte message whose destination is the entire MIDI system. An example is a message to request tuning.

A system real time message is a 1 byte message whose destination is the entire MIDI system and whose purpose is synchronization. An example is a MIDI clock message.

A system exclusive message is a variable length message whose destination is a specific MIDI device. An example is a message to change the voice (patch) of a synthesizer.

1.3 Classes for MIDI Data

In C++, a stream is an abstraction for the flow of data. In the design of classes for manipulating MIDI data, the MIDI stream is called mstream and is similar in usage to the C++ file stream called fstream and the C++ standard I/O stream called iostream. In addition to mstream, there are the following supporting classes:

ICMC 459
2. Low Level Control

In several ways, MIDI is analogous to ASCII. For example, it is sometimes convenient to process a single ASCII character at a time. Similarly, in MIDI applications it is sometimes convenient to process a single MIDI message at a time. To provide this low level control, message queues are configured as shown in Figure 1.

2.1 Low Level Input

The paths to the queues are enabled by the `"=""` operator and disabled by the `!=""` operator. For example, the following enables the path from mstream midi's input to its queue for channel messages:

```c
midi = = midi.charQueue;
```

Once enabled, the `"*"` and `"/="` operators are used to determine if the queue has received any messages. For example, mstream midi's channel queue is not empty if (`"(*midi.charQueue !=" 0)"`) is true. In order to remove a message from a queue, the `">>"` input operator must be used. For example, the following receives channel msg from mstream midi's channel queue:

```c
midi >> msg;
```

2.2 Low Level Output

Unlike the input, the output has no message queuing. Therefore, no queue enabling operations are necessary. The operator for mstream output is `"<<"`. For example, the following transmits channel msg to mstream midi:

```c
midi << msg;
```

2.3 Low Level By-pass

Quite often, MIDI applications are required to echo all MIDI input back to MIDI output. Therefore, mstreams have a by-pass path. As with the paths to the queues, the by-pass path is enabled with the `"="` operator and disabled with the `!="` operator. For example,

```c
midi = = midi;
```

enables mstream midi's by-pass path.
3. Mid Level Control

Continuing with the MIDI/ASCII analogy, it is often more convenient to process variable length arrays of ASCII characters, called strings, than processing a single character at a time. Similarly, it is often more convenient to process variable length sequences of MIDI messages, called Tracks, than processing a single MIDI message at a time.

A Track is just a storage object. In order to produce sound, a Track must be read and played, which is done by an object called a Player. A Score is an object for storing a group of Tracks. Finally, an Ensemble is a group of Players which read and play a Score.

3.1 Mid Level Input

The operator for the input of Tracks and Scores is ">>", which can be used to input MIDI data from a MIDI stream (mstream) or a file stream (fstream). In C++, a stream's function that does formatting is called a manipulator. The MIDI library provides the following manipulators for MIDI data:

- smf, for formatting MIDI data in Standard MIDI File format (2)
- text, for an ASCII representation of the MIDI data
- data, for a binary representation of the MIDI data

For example, the following input Score s
from stream fn, which contains a Standard MIDI File:

```c++
fn >> smf >> s;
```

3.2 Mid Level Output

The operator for the output of Tracks and Scores is "< <", which can be used to output MIDI data to an fstream or an mstream. Of course, the three manipulators used with the ">>" input operator may also be used with the "<<" output operator.

As previously mentioned, MIDI data from a Track or Score must be read and played by a Player or Ensemble before it can be output to an mstream. Therefore, it is necessary to assign a Track or Score to a Player or Ensemble. This assignment is done by the "->" operator. For example, the following assigns Score s to Ensemble e, which then plays the Score to mstream midi:

```c++
  s > e;
  midi < < e;
```

Before or during playing, the following operations control Players:

- | n, changes output to channel n
- | n, transposes n half steps
- > n, increases note-on velocity by n
- < n, increases note-off velocity by n
- >> n, delays by n time units
- << n, scales time by n
- % n, quantizes time to n time units

3.3 Mid Level Example

The following example of the use of mid level control records a Track and creates and plays an Ensemble which contains the originally recorded Track along with its echoes. The example allows for the user specification of the echo time delay, volume reduction, and transposition. Note that many Players are using the same Track.

```c++

/*
 * rec_play.cpp
 * Record a Track, then play it with echoes.
 * Binds to MIDI and LINK: Binded C++:
 * bcc rec_play.cpp mids.lib
 * ARGUMENTS:
 * (int) # echoes
 * (int) # ms delay between Track pointers
 * (int) echo volume reduction
 * (int) echo transposition
 * PROCESSING:
 * Read command line.
 * ICMI 461
 */
```
Record until any key is entered.
Prepare Player(0).
Play.

```c
#include <mstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[]) {
    int i, copies, delay, volume, trans;
    Player *p;
    Ensemble e;
    Track t;

    // Read command line:
    if (argc != 5) {
        copies = atoi(argv[1]);
        delay = atoi(argv[2]);
        volume = atoi(argv[3]);
        trans = atoi(argv[4]);
    } else {
        cerr << "usage : rec_play count delay volume transpose" << " exit(1);"
    }

    // Record until any key is entered.
    midi >> > t;
    while (!bkbhit()); getc(); midi.stop();

    // Prepare Player(s).
    for (i = 0; i < copies; i++) {
        // Assign Track to new Player.
        p = new Player;
        t = *p;

        // Add to Ensemble a Player delayed,
        // volume controlled, and transposed.
        c = *p > > t.delay > > volume > > t.trans;
    }

    // Play.
    midi << e;
    while (midi.output());
```

4. Conclusion

C++ is a good fit for MIDI data manipulation because

- it is efficient enough for real-time applications
- object oriented tools are easy to learn when they are designed to mirror real world objects already familiar to the user
- operator overloading makes source code more symbolic and easier to read and write.

The described C++ class design for low and mid level MIDI data manipulation has been implemented in version 2 of the cmIDId C++ class library, which currently requires an IBM PC/AT/PS2, a Roland MPU-401 interface, DQX, a Microsoft or Borland C++ compiler, and a Microsoft or Borland assembler to rebuild library. The library (written by Sepes) is available from cmIDId.

Although the MIDI/ASCII analogy is difficult to stretch farther, work is being done on a higher level of MIDI data manipulation. This higher level has the ability to dynamically alter, based upon Track content and user input, the ordering of a list of Tracks being played. Work is also being done on porting the cmIDId class library to the Next computer.

5. References

[1] "MIDI 1.0 Specification", The International MIDI Association, 5316 W.
57th Street, LA, CA 90056, (213)649-6434

57th Street, LA, CA 90056, (213)649-6434.


ICMC 462