

Overview.

In the 1920s, composer Arnold Schoenberg devised a method for music generation known as *serialism*. In its most basic form, this method is easy to program. Although the music resulting from a basic implementation is unlikely to impress, it serves as a good introduction to the techniques needed for structuring slightly larger programs.

The assignment consist of 2 parts: the coding & designing and the documentation

Your task is to implement a monophonic serial music generation program that:

- a) Takes input from a file (in the format described below)
- b) Generates a randomized serial composition based on this input.
- c) Exports the composition in a standard output format (also described below).

As with any nontrivial programming task, it is necessary to become familiar with the vocabulary of the *problem domain* (see Booch's "Object-Oriented Analysis and Design" or "C++ FAQs" for more details). For this coursework, this involves a minimal quantity of musical terminology. However, no particular grasp of music theory or musical ability is assumed or required.

Basic Method for Serial Music Generation.

A serial composition is built upon the concept of a *tone row*: a sequence of notes that are used as a basis for generating musical phrases (e.g. [C,D#,G]).

Each of the 12 notes in the list [C, C#, D, D#, E, F, F#, G, G#, A, A#, B] has an associated integer value in the range 0-11 (inclusive) given by the *index* of the note in the list.

A *phrase* is created from the notes of a tone row by additionally specifying for each individual note:

- a) A *velocity*, corresponding to the loudness of the note (an integer value in the range 0-127 inclusive, with 0 being silent).
- b) An *octave* (an integer value in the range 0-7 inclusive).
- c) An integer *duration* (more details on this in the section on 'output format', below).

New phrases can be generated from existing phrases by the following operations:

- a) Transposition. A note can be increased (or decreased) in value by an integer number of semitones in the range 1 to 11 (inclusive). Thus, C transposed by 1 semitone becomes C# etc. Transposition is modulo 12, so C transposed by 11 semitones becomes B etc.
- b) Retrograde. A series of notes (or velocities, durations etc) can be reversed. Hence [C,D#,G] becomes [G,D#,C].
- c) Inversion. The inversion of a note with an index of i is the note with index: $12 - i - 1$
- d) Retrograde-inversion. A combination of b) and c).

A serial composition is then obtained by applying the above operations to a sequence of *base phrases*, in order to create a sequence of generated phrases.

Technical specification.

1. Input.

Command line:

The program takes two command-line parameters:

- a) the pathname of a file containing the base phrases
- b) a string that determines the composition strategy (as detailed below)

Keyboard:

The program takes no keyboard input.

2. Output.

The output will be to a text file as described below.

3. Operation.

A. Parse the base phrases.

The program should be supplied with a command line parameter giving the name of a file that contains *one or more* base phrases.

The EBNF for the phrase file format is as follows:

```
<input_file> ::= <phrase> { <phrase> }
<phrase> ::=
<"tonerow"> <ws> <note> { <ws> <note> } <cr>
<"durations"> <ws> <duration_value> { <ws> <duration_value> } <cr>
<"velocities"> <ws> <velocity_value> { <ws> <velocity_value> } <cr>
<"octaves"> <ws> <octave_value> { <ws> <octave_value> } <cr>
<ws> ::= whitespace - one or more of space (' ') or tab ('\t') (but not '\n')
<cr> ::= carriage-return/linefeed - equivalent to the '\n' character.
<note> ::= "C" | "C#" | "D" | "D#" | "E" | "F" | "F#" | "G" | "G#" | "A" |
"A#" | "B"
<duration_value> ::= integer in the range 0-65535 (inclusive)
<velocity_value> ::= integer in the range 0-127 (inclusive)
<octave_value> ::= integer in the range 0-7 (inclusive)
```

The associated semantics are that there are the same number of durations, velocities and octaves in a phrase as there are notes.

Example:

```
tonerow A B C# D
durations 30 30 30 30
velocities 60 70 80 100
octaves 2 1 2 1
```

B. Generate a new Composition.

The second command line parameter governs the choice of composition strategy.

There are two basic strategies required: "identity" and "random", described as follows:

"identity": this strategy just echoes the basic phrases, in the same order as they appear in the input file.

"random": this strategy operates by repeatedly choosing a random basic phrase and then applies one of the above operations (also chosen at random) to generate a new phrase.

NB: In terms of meeting the strict requirements of this coursework, the above two strategies are all that is required.

The primary concern for marking purposes is the quality of your design and coding, rather than the "musical quality" of the output. However, investment in attempting to make your program produce more "pleasing" or "interesting" output is likely to have benefits that may ultimately reflect in improved marks. The reasons for this include (but are not limited to):

- The extra time you spend working with your code is likely to help uncover bugs.
- Attempting to use existing functionality in a slightly different context may suggest interface and/or implementation improvements.
- Overall, it will give your greater familiarity with the *evolutionary* aspect of the software lifecycle (the most-neglected, but most important part).

There are any number of things that one might do to make the generation strategy produce more "interesting" output. Here are just a couple of ideas:

a) A "permutation" operation could be added to reorder the elements of a sequence (of notes, octaves etc). Recall that a sequence with n elements has $n!$ permutations. Retrograde is of course just one particular permutation, but it is given special status in serialism because it is a musically common technique.

In the minimalist opera "*Akhmaten*", the composer Philip Glass uses permutations of velocities of short duration notes to achieve a hypnotic effect.

b) A "generate and test" strategy could be applied to the choice of operation in order to make successive phrases "flow" together more evenly. One definition of "flow" might be that the difference in note values between the end of one phrase and the start of the next is a minimum. *Any additional strategies should appear prominently in your documentation, together with the command-line string necessary to select them* (perhaps also with a suggestion as to your favoured strategy and why this is so).

**NB: In the interest of fairness to the coursework markers:
Generated output should be less than a minute in length.
There should be no more than a small number of additional strategies.**

C. Output a Composition.

After generation, a composition should be output in the format expected by the **t2mf** program (docs and executable for this in the Coursework 1 folder on Web-CT) to the file "serialism.txt". An example of the basic format (which consists of a single middle 'C' of quarter note duration) is as follows:

```
MFile 0 1 96
MTrk
0 TimeSig 4/4 24 8
0 Tempo 500000
0 On ch=1 n=36 v=60
96 Off ch=1 n=36 v=60
96 Meta TrkEnd
TrkEnd
```

To adapt this example for your output purposes, *the only thing that needs to change* is that you must output a pair of On .. Off events for *each* generated note.

The On..Off entries are of the form:

<timestamp> <On|Off> <channel> <note> <velocity>

The meaning of these is as follows:

<timestamp>

This is the time (in clock ticks) of event. The value of 96 on the first line is the *timebase* (in clockticks- per-quarter note). There is no requirement to use anything other than quarter notes, in which case the durations of the notes in the input base phrases (i.e. the difference between the number at the start of the On line for a note and the corresponding Off line) will be an integer multiple of the timebase value.

<channel>

This should always be one.

<note>

Middle 'C' (octave 3, index 0 in the input format) has value 36. Hence a note as specified by the pair (octave,index) in the input format is output as note value (12 * octave) + index.

<velocity>

This is the same as velocity in the input format. Note that a velocity of 0 can be used to introduce a rest into the composition.

Note: If your composition sounds like it's ending prematurely, make sure that the TrkEnd marker is actually the last thing to appear (in terms of timing).

The t2mf program is can then be used to convert the "serialism.txt" file into a MIDI (.mid) file that can be played by (for example) Windows Media Player.

Note that t2mf seems to require *piped* input, i.e. use it with the command line:

t2mf < serialism.txt > serialism.mid

General Implementation Issues.

- You may not make use of any third-party libraries (e.g. boost/stlport etc).
- Your program must not make any direct operating system calls (via std::system() or otherwise).
- With the exception of your own header files, your program should only make use of those provided by the standard c++ library. For completeness, these are:
<algorithm> <bitset> <complex> <deque> <exception> <fstream> <functional> <iomanip> <ios>
<iosfwd> <iostream> <istream> <iterator> <limits> <list> <locale> <map> <memory> <new>
<numeric> <ostream> <queue> <set> <slist> <sstream> <stack> <stdexcept> <streambuf>
<string> <stringstream> <typeinfo> <utility> <valarray> <vector> <cassert> <cctype> <cerrno>
<cfloat> <ciso646> <climits> <locale> <cmath> <csetjmp> <csignal> <cstdarg> <cstddef>
<cstdio> <cstdlib> <cstring> <ctime> <wchar> <wctype>

- **A good design will attempt to minimize the degree of coupling between components (i.e. high modularity) and maximize the amount of information hiding (i.e. high encapsulation).**

Depending on the problem, the components will be procedural, object-based or object-oriented. This coursework lends itself well to an object-based decomposition. The first step is therefore to

find the objects (see Booch or any other good OOA/D book for more on this). Here, a 'natural' decomposition of the specification into objects should be fairly easy to find. By 'natural', we mean one in which the division of responsibility between classes is clear-cut and encapsulation is maximized.

More specifically:

The main components for decomposition should be classes.

Your classes should not have public attributes.

There should be absolutely no 'global' variables.

For our purposes, 'Global' variables are variables that are neither local nor attributes of a class. Having one or two global *constants* (to avoid magic numbers, e.g. `const int notes_per_octave = 12;`) is OK (though they would best be static class attributes).

The design should be outlined in the documentation and the overall structure of this design should be evident at the topmost level of your code (i.e. without requiring detailed inspection).

Clarity of Implementation

Write your code so as to make it easy for other people to read it.

- Decompose functionality in a modular fashion.
- Decompose modules into functions and/or classes as appropriate.
- Have as few points of communication between modules as possible.
- Keep functions/methods short. If a function is more than a few dozen lines long, there's generally a good way of splitting it up to make it clearer.
- Each class or function preferably serve a single well-defined (and preferably documented) purpose.
- Evidence of error-handling (such as widespread use of design by contract, preferably supported by the use of `assert`) will go a long way to convince readers of your code that it will always behave as it should.

Documentation.

Provide a `readme.txt` file containing the following:

1. Installation, compilation and execution instructions.
2. Areas of Personal Interpretation (i.e. how your program differs in areas where you believe the spec is ambiguous).
3. Known limitations (i.e. bugs). Note that submissions that are less than honest about their limitations will receive reduced marks.
4. High-level program design.

This should be at a much higher level than saying that you used a 'for' loop etc. Broadly speaking, this might include:

- a). How the original problem was split up into subproblems.
- b). How the solutions to these subproblems are (very broadly) implemented.
- c). How these solutions communicate with each other.

In general, it should be possible to convey the overall structure of a program without going into low-level detail.

NB: Very important submission

You must submit the following files:

- Your project source code.
- A Dev-C++ project to compile your code.
- Full documentation in the form of a `readme.txt`

All of these are vital. Make sure that they are all included.

If you submit code that does not compile under **Dev-C++**, then you will receive minimal marks - very close to zero!