# Ableton Programming Tasks

We created some programming tasks to choose from. They are all somewhat related to our product Ableton Live, although we tried to make the descriptions self-contained so that no in-depth knowledge of Ableton Live is required.

Your solution should meet the following requirements:

- Please use Standard C++.
- Additionally, you may use Google Test. Please use no further libraries.
- Please submit a Makefile, Xcode project, or a Visual Studio solution if necessary.
- The resulting command line application reads input as described in the individual tasks from standard input. No extra input (e.g. command line options) should be necessary to successfully run the application.
- All output is written to standard output and adheres to the spec in the individual tasks and does not contain additional information.

You can assume that your program is only run with inputs that comply to the specifications in the individual tasks.

Please craft your solution in the same way you would write production code.

Please choose one task that is appropriate for you. Completing task 1, 2, or 3 should take about two hours, whereas completing task 4 should take about four hours.

## Task 1: Musicians don’t count from zero (easy).

Musical time (Beat Time) is often displayed in Live’s user interface. We use a notation that consists of 3 parts separated by dots. The first part denotes the bar, the second the beat (somehow the metronome tick within the bar) and the last part the 16th note relative to the beat. For example, in 4/4 time the string “3.2.4” denotes the 4th 16th note of the second beat of the 3rd bar, i.e. one 16th before the middle of the bar. Beat times that do not match a full 16th note are indicated by an appended “+” sign, e.g. “2.1.1+” would indicate any beat time between “2.1.1” and “2.1.2”.

However, internally we just use doubles to represent the beat time. The integral part of the double refers to the bar (zero based).

We want you to write a conversion between beat time in doubles and the above mentioned format.

Input description

First line: Number of Beats per Bar (Integer)
Second line: Number of 16th notes per Beat (Integer)
Remaining lines: Beat times to be converted (one double per line).

Output description

One line for each converted beat time containing the time to be converted followed by a space and the result of the conversion.

Example

Input

```
4
4
1.55
```

Output

```
1.55 2.3.1+
```

## Task 2: It's all warped (medium).

In Ableton Live, Warp Markers can be used to pin a certain beat time (position in musical time) to a certain sample time, i.e. position in the audio material. They are used to tell Live's engine at which tempo (= beat time / sample time) a given piece of material should be played.

Technically, a set of Warp Markers is part of each Clip in Live. They can be generated by analyzing the audio or be adjusted manually by the user. A Clip can contain an arbitrary number of Warp Markers.

If you are curious, Warp Markers are described in detail in section 9.2.2 of Live's user manual. But for the sake of this task just assume the following behavior:

- There is at least one Warp Marker in the clip.
- Between two Warp Markers, the tempo is constant.
- The tempo before the first Warp Marker is the same as the tempo after the first Warp Marker.
- The tempo after the last Warp Marker is specified separately in the input.

Input description

There are 4 kinds of lines in the input:

1. Warp Marker definition.
2. Definition of the tempo after the last marker.
3. Sample time to beat time conversion.
4. Beat time to sample time conversion.

They can appear in any order, but at least one Warp Marker and the tempo after the last Warp Marker will be defined before the first conversion.
Each line consists of a keyword followed by numeric arguments. All times are given as doubles in arbitrary units. The tempo after the last Warp Marker is given as beat time/sample time in the same units. Warp Marker and tempo definitions affect only the conversions that come later in the input.

```
marker <beat time> <sample time>
end_tempo <value>
s2b <sample time>
b2s <beat time>
```

Output description

For each of the “s2b” and “b2s” lines the corresponding output time is printed.

Example

Input

```
marker 0.0 0.0
marker 1.0 5.0
end_tempo 10.0
b2s 0.5
s2b 6.0
```

Output

```
2.5
11.0
```

## Task 3: Follow me (medium).

A fun feature of Live are the so-called “follow actions”. For a Clip (that is either a piece of audio or MIDI material) users can set a property that tells Live that it should start another clip automatically when this clip has been playing for a certain amount of time. There is also some randomness in the decision which clip should ‘follow’. Therefore, the feature can be used for some kind of algorithmic composition.

We want you to write a program that simulates this behavior in a simplified way. The input will contain ‘clip definitions’ with follow actions for each clip and also ‘ticks’ that drive the playback, i.e. with every tick message the ‘playback’ should advance one tick. The first tick ‘starts’ the first clip. The clip after the last clip is regarded the first clip and vice versa.

Input description
There are two types of input lines, ‘clip’ (with space separated parameters) and ‘tick’.

```
clip <name> <ticks_to_play> <follow_chance1> <follow_chance2> <action1> <action2>
```

Each clip line defines an ‘audio clip’ together with its follow actions: ticks_to_play is the number of ticks that the clip should ‘play’ before it triggers another clip as defined by the follow actions. There are two actions associated with the clip, they are chosen randomly so that their relative occurrence corresponds to the given ‘chances’, e.g. follow_chance1 = 2 and follow_chance2 = 3 implies that action2 is executed (on average) 1.5 times as often as action1. Possible values for action1 and action2 are

- `none` : nothing happens, the clip keeps playing
- `any` : any clip (including the playing one) can be triggered
- `other` : any other clip can be triggered
- `next` : the next clip (in the order they were created) will be triggered
- `previous` : the previous clip will be triggered

If a clip with the same name is created again, it replaces the already existing one.

The second type of input line just consists of the word `ticks` followed by the number of ticks to process and optionally two ‘random numbers’ between 0 and 1 for each tick. The random numbers are given in the input so that the output is deterministic, i.e. allows us to run a test input against different implementations. If this input is absent, real random numbers should be used. (It’s part of the task to find out why there are two random numbers needed for each tick).

Output description
For each tick a single output line is created containing only the name of the currently playing clip.

Example

Input

```
clip hello 2 1.0 0.0 next none
clip world 3 1.0 0.0 previous none
ticks 10
```

Output

```
hello
hello
world
world
world
hello
hello
world
world
world
```

## Task 4: Modular Madness (hardest by far).

In audio software, sound processing is done using a network of modules, each executing a simple task like applying a filter or summing signals. In this task we process strings instead of audio using a similar module-based approach.

The input defines a network of modules and a stream of input into this network. This input stream should be fed into the first module (in the order of definition) and the output of the program should correspond to the output of the last module. If there are multiple input connections for a module, they should be 'summed' before feeding to the module. Summing works by appending the strings in the order in which the input connections have been made. The individual modules should be processed in the order they have been defined. The network should only process one string at a time. Each 'process' statement should let the network 'run empty' (see also the example below).

If you encounter ambiguities while solving this task make an assumption about the behavior of the network and document it in an appropriate way.

Input description

Each line starting with the keyword `module` defines a new module to be added to the network. A line starting with the keyword `connect` connects the output of one module to the input of another module. A line starting with `process` feeds the rest of the line into the first module. This input consists of one or more strings made of the characters a-z. Individual strings are separated with a single space character. If there are several `process` lines each of them should behave the same way as if input alone.

A module definition looks like this:

```
module  <name> <operation>
```

Here name is an arbitrary name (no whitespace) for the module and operation specifies what the module should do with its input. The following operations should be supported:

- `echo` : The output string is the input string concatenated to itself.
- `reverse` : The output string is the input string reversed.
- `delay`: The output string is the previous input string. The initial output is "hello".
- `noop`: The input appears unchanged at the output.

A connection is made like this

`connect <module_name1> <module_name2>`

This connects the output of module `module_name1` to the input of module `module_name2`. Modules will always be defined before they are connected.

Output description

For each “process” line in the input an output line should be created that contains the outputs of the last module separated by a single space character. The output should be limited to sixteen times the number of strings of the input.

Example
Input

```
module alpha reverse
module beta delay
connect alpha beta
process hello world
```

Output

```
hello olleh dlrow
```