

# CS 3005: Programming in C++

## Envelope Classes

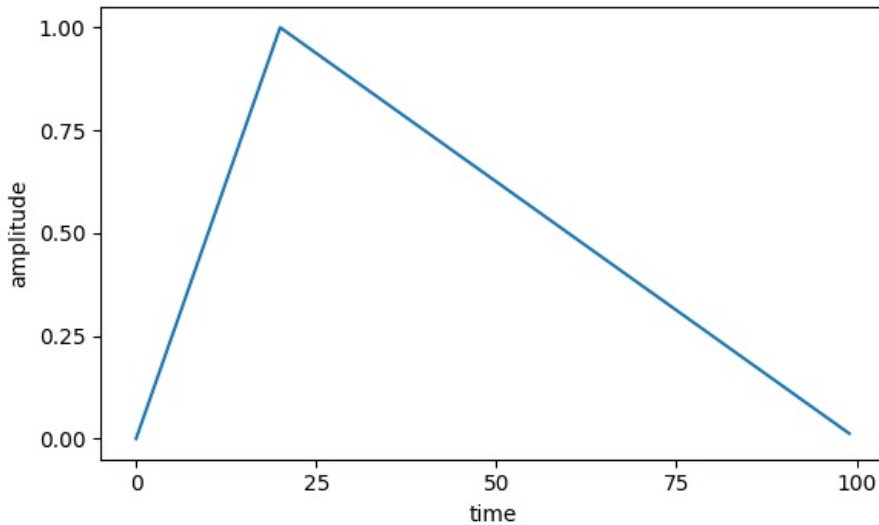
### Introduction

Envelopes are used in audio synthesis to control the volume of a sound over time. For example, a sound may start out loud, then become quieter over time. Or it may take time to reach full volume.

Audio designers will try many different envelope shapes to create different sounds.

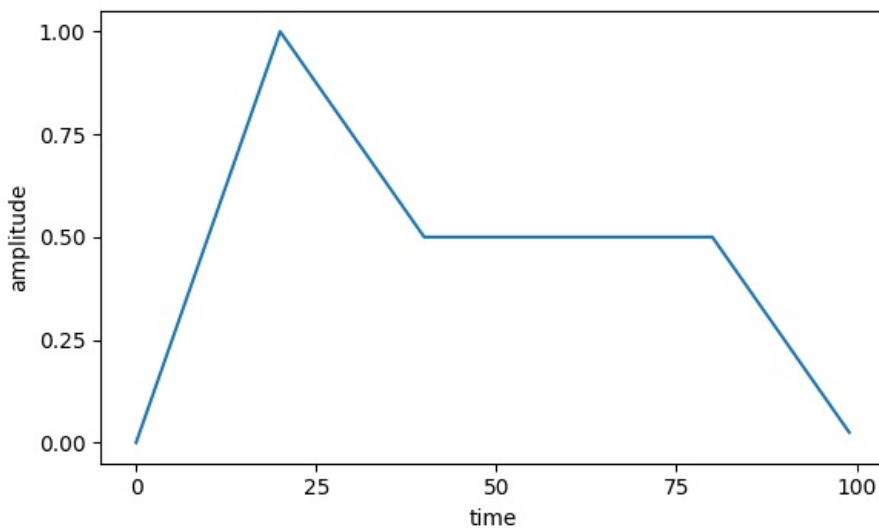
A simple envelope shape is the “Attack/Decay” (AD) envelope. In this envelope style, the volume initially increases from 0 to the maximum. It then decays to 0.

Envelope



A common envelope shape is the “Attack/Decay/Sustain/Release” (ADSR) envelope. In this envelope style, the volume initially increases from 0 to the maximum. It then decays to an intermediate (sustain) level. Next, it stays at this level for some sustain duration. Finally, it decays to 0 (release).

Envelope



You may notice that the AD envelope is a simplified form of the ADSR envelope.

### Assignment

In this assignment, you will create a class hierarchy to represent the functionality of an envelope, including

specialization for the ADSR and AD envelopes.

The `Envelope` class will be a base class. `ADSREnvelope` will inherit from `Envelope`. `ADEnvelope` will inherit from `ADSREnvelope`.

You will also create a simple program that generates an envelope and displays its amplitudes as text. In a future assignment, you'll use the envelopes together with waveforms to generate sounds.

A sample interaction with the program may look like this:

```
$ ./program-envelope-test/envelope_test
Samples/Second: 10
Seconds: 1.0
Envelope style: ADSR
Maximum amplitude: 0.75
Attack seconds: 0.2
Decay seconds: 0.3
Release seconds: 0.4
Sustain amplitude: 0.25

sample_number,amplitude
0,0
1,0.375
2,0.75
3,0.583333
4,0.416667
5,0.25
6,0.25
7,0.1875
8,0.125
9,0.0625
```

Notice that the output is in CSV form, so you could plot it to produce images like shown above.

This example demonstrates a user giving an invalid envelope style.

```
$ ./program-envelope-test/envelope_test
Samples/Second: 10
Seconds: 1.0
Envelope style: USPS
Maximum amplitude: 0.8
Envelope style 'USPS' is not known.

sample_number,amplitude
0,0
1,0
2,0
3,0
4,0
5,0
6,0
7,0
8,0
9,0
```

## Computing Envelopes

Let's discuss the computational steps for the ADSR envelope. Each envelope computation has a duration in seconds. We'll use `t` to denote duration. Each envelope computation also has a samples per seconds. We'll use `sps` to denote this value. If you multiply these two values, you get the number of samples that will be computed by the envelope. We'll use `n` to denote the number of samples. Let `a` be the attack duration in seconds, `d` be the decay duration in seconds, `s` be the sustain duration in seconds, and `r` be the release duration in seconds.

In order to compute the amplitudes for the ADSR envelope, we need to compute straight lines in 4 different sections, with each section connecting to the one before and one after it.

### Attack Section

The attack section starts at amplitude 0.0 and ends at the maximum amplitude. The sample positions start at

position 0, and end after the attack duration. The position of the end of the attack duration can be computed by  $\text{sps} * a$ .

## Decay Section

The decay section starts at the maximum amplitude and ends at the sustain amplitude. The sample positions start at position where the attack ends, and end after the decay duration. The position of the end of the decay duration can be computed by  $\text{attack\_end} + \text{sps} * d$ .

## Sustain Section

The sustain section starts at the sustain amplitude and ends at the sustain amplitude. The sample positions start at position where the decay ends, and end after the sustain duration. The sustain duration can be computed as  $s = t - (a+d+r)$ .

The position of the end of the sustain duration can be computed by  $\text{release\_end} - \text{sps} * r$ .

## Release Section

The release section starts at the sustain amplitude and ends at 0.0 amplitude. The sample positions start at position where the sustain ends, and end at the end of the envelope. The position of the end of the release duration can be computed by  $t * \text{sps}$ .

## Computing “Straight Line” Amplitudes

Each of the sections above need to compute a straight line of amplitudes for their sections. Each as a starting amplitude ( $y_0$ ), an ending amplitude ( $y_1$ ), a starting position ( $x_0$ ) and an ending position ( $x_1$ ). Let  $x$  be the position we want to compute for, and  $y$  be the amplitude we want to compute.

We'll use the formula from algebra for computing straight lines from two points.

$$y = (x - x_0) * (y_1 - y_0) / (x_1 - x_0) + y_0$$

## Programming Requirements

**Create** `library-envelope/Envelope.{h,cpp}`

### Envelope Class

#### Data Members:

The `Envelope` class should contain data members to track the following information. These data members should be `protected` or `private`. They are not allowed to be `public`.

- `std::string` name for the envelope; Used to identify an envelope in collections of many envelopes.
- `std::string` type name for envelope; Used to identify the specialized form of the envelope, such as “AD” or “ADSR”.
- `double` maximum amplitude; The maximum amplitude of the envelope.

#### public Methods:

- `Envelope(const std::string& name, const std::string& type_name);` Constructor for the class. Initializes two data members from the input parameters, and sets maximum amplitude to 1.
- `Envelope(const std::string& name, const std::string& type_name, const double amplitude);` Constructor for the class. Initializes all data members from input parameters.
- `virtual ~Envelope();` Empty body. Required for polymorphic class.
- `virtual void generateAmplitudes(const double seconds, const int samples_per_second, AudioTrack& track) const = 0;` Pure virtual method. Will be overridden by subclasses to generate amplitude values specific to envelope form.
- `const std::string& getName() const;` Returns the data member.
- `const std::string& getTypeName() const;` Returns the data member.
- `double getMaximumAmplitude() const;` Returns the data member.
- `void setName(const std::string& name);` Sets the data member from the parameter.
- `void setType_name(const std::string& type_name);` Sets the data member from the parameter.

- `void setMaximumAmplitude(const double amplitude);` If `amplitude` is greater than 0, sets the maximum amplitude data member.

## Create `library-envelope/ADSREnvelope.{h,cpp}`

### `ADSREnvelope` Class

Publicly inherits from `Envelope`.

### Data Members:

The `ADSREnvelope` class should contain data members to track the following information. These data members should be `protected` or `private`. They are not allowed to be `public`.

- `double` attack duration in seconds; See discussion above for more details.
- `double` decay duration in seconds; See discussion above for more details.
- `double` sustain amplitude; See discussion above for more details.
- `double` release duration in seconds; See discussion above for more details.

### `public` Methods:

- `ADSREnvelope(const std::string& name, const std::string& type_name);` Constructor chains, passing these parameters to the parent class. Initializes the data members for attack, decay, sustain, and release to 0.0, 0.0, 0.5 and 0.0, respectively.
- `ADSREnvelope(const std::string& name, const std::string& type_name, const double maximum_amplitude, const double attack_seconds, const double decay_seconds, const double sustain_amplitude, const double release_seconds);` Passes the first three parameters to the parent class via constructor chaining. Uses the rest of the parameters to initialize the data members for attack, decay, sustain, and release.
- `ADSREnvelope(const std::string& name);` Constructor chains, passing the parameter to the parent class. Also passes "ADSR" as the second parameter to the parent constructor. Initializes the data members for attack, decay, sustain, and release to 0.0, 0.0, 0.5 and 0.0, respectively.
- `ADSREnvelope(const std::string& name, const double maximum_amplitude, const double attack_seconds, const double decay_seconds, const double sustain_amplitude, const double release_seconds);` Passes the first two parameters to the parent class via constructor chaining. Also passes "ADSR" as the second parameter to the parent constructor. Uses the rest of the parameters to initialize the data members for attack, decay, sustain, and release.
- `virtual ~ADSREnvelope();` Empty body. Required for polymorphic inheritance.
- `double getAttackSeconds() const;` Returns the data member.
- `double getDecaySeconds() const;` Returns the data member.
- `double getSustainAmplitude() const;` Returns the data member.
- `double getReleaseSeconds() const;` Returns the data member.
- `void setAttackSeconds(const double attack_seconds);` If the parameter is greater than 0, sets the data member.
- `void setDecaySeconds(const double decay_seconds);` If the parameter is greater than 0, sets the data member.
- `void setSustainAmplitude(const double sustain_amplitude);` If the parameter is greater than 0, sets the data member.
- `void setReleaseSeconds(const double release_seconds);` If the parameter is greater than 0, sets the data member.
- `virtual void generateAmplitudes(const double seconds, const int samples_per_second, AudioTrack& track) const;` If the `seconds` parameter is less than the sum of the attack, decay, and release durations, does nothing. Otherwise, sets the track size using the other parameters. It then computes the positions in the track for the transitions between attack, decay, sustain, and release. It then uses the other methods to assign the amplitudes to the track.
- `void assignAttackAmplitudes(const int begin, const int end, AudioTrack& track, const double a0, const double a1) const;` The positions in track starting at `begin` and ending before `end` are updated to be a linear ramp from `a0` to `a1`.
- `void assignDecayAmplitudes(const int begin, const int end, AudioTrack& track, const double a0, const double a1) const;` The positions in track starting at `begin` and ending before `end` are updated to be a linear ramp from `a0` to `a1`.
- `void assignSustainAmplitudes(const int begin, const int end, AudioTrack& track, const double a0) const;` The positions in track starting at `begin` and ending before `end` are updated to be `a0`.
- `void assignReleaseAmplitudes(const int begin, const int end, AudioTrack& track, const double a0, const double a1) const;` The positions in track starting at `begin` and ending before `end` are updated to be a linear ramp from `a0` to `a1`.

**Create** `library-envelope/ADEnvelope.{h,cpp}`

## `ADEnvelope` Class

Publicly inherits from `ADSREnvelope`.

### Data Members:

The `ADEnvelope` class does not define any new data members.

### `public` Methods:

- `ADEnvelope(const std::string& name);` Passes the first parameter to the parent class via constructor chaining. Also passes the values “AD”, 1.0, 0.0, 0.0, 0.0, and 0.0.
- `ADEnvelope(const std::string& name, const double maximum_amplitude, const double attack_seconds);` Constructor chains to the parent class, using the parameters here. Also uses “AD” for the type name, and 0.0 for all other unspecified values.
- `virtual ~ADEnvelope();` Empty body, but required.
- `virtual void generateAmplitudes(const double seconds, const int samples_per_second, AudioTrack& track) const;` If the `seconds` passed as a parameter does not allow for the attack duration to complete, does nothing. Otherwise fills the track data with an attack and a decay stage. Uses the `assignAttackAmplitudes()` and `assignDecayAmplitudes()` methods.

**Create** `library-envelope/Makefile`

This file must contain rules such that any of the following commands will build the `libenvelope.a` library:

- `make`
- `make all`

This file must contain rules such that the following command will install the `libenvelope.a` library into the `lib` directory, and all `.h` files to the `include` directory:

- `make install`

**Update** `library-commands/wav_file_creator_aux.{h,cpp}`

### Functions:

- `void configure_audio_track(ApplicationData& app_data);` This function configures the `seconds` and `samples per second` meta data for the `AudioTrack` object in the `ApplicationData` object. It prompts the user for the integer “Samples/Second: ” and the double “Seconds: ”, then stores them in the `AudioTrack` object.

**Create** `library-commands/envelope_test_aux.{h,cpp}`

### Functions:

- `void fill_audio_track_with_envelope(ApplicationData& app_data);` Prompts the user for string “Envelope style: ” and double “Maximum amplitude: “. Create an envelope object based on the user’s choices. If the user chose `ADSR`, then prompt the user for the doubles “Attack seconds: “, “Decay seconds: “, “Release seconds: “, and “Sustain amplitude: “. Use the values to configure the meta data of the envelope. If the user chose `AD` then prompt the user for the double “Attack seconds: “. Use the value to configure the meta data of the envelope. If the user chose anything else for the envelope style, give an error message “Envelope style ‘USER\_CHOICE’ is not known.” If the choice was one of the allowed ones, set the envelope’s maximum amplitude, then fill the `AudioTrack` object using `generateAmplitudes()` from the envelope object. Note that the parameters needed by this method are in the `AudioTrack`’s meta data.
- `int envelope_test(ApplicationData& app_data);` Uses `configure_audio_track()` to configure the `AudioTrack` object in the `ApplicationData` object with the `seconds` and `samples per second` meta data. Uses `fill_audio_track_with_envelope()` to fill the `AudioTrack` object with envelope data, according to the user’s choices. Uses `display_audio_track()` to send the `AudioTrack` object to the text output. Returns 0.

**Update** `library-commands/Makefile`

Add `envelope_test_aux.{h,cpp}` in the appropriate places to add them to the library and install the header file.

**Create** `program-envelope-test/envelope_test.cpp`

## Functions:

- `int main();` Entry point to the envelope test program. Should create an `ApplicationData` and pass it to the `envelope_test` function found in `envelope_test_aux` and return the result of that function call.

**Create** `program-envelope-test/Makefile`

This file must contain rules such that any of the following commands will build the `envelope_test` program:

- `make`
- `make all`
- `make envelope_test`

**Create** `program-envelope-test/.gitignore`

The file `program-envelope-test/.gitignore` needs to store one line of text:

```
envelope_test
```

**Update** `Makefile`

Update the project-level `Makefile` so that `make` and `make all` in the project directory will call `make install` in the `library-envelope` directory, and `make` in the `program-envelope-test` directory.

## Additional Documentation

- [Detailed ADSR Example](#)

## Grading Instructions

To receive credit for this assignment:

- your code must be pushed to your repository for this class on GitHub
- all unit tests must pass
- all acceptance tests must pass
- all programs must build, run, and execute as described in the assignment descriptions.

## Extra Challenges (Not Required)

TBA