Evo Engine

Project Setup

I'm going to assume some knowledge of basic CMake or C++ project structure and pick a solid starting point for our project. The CMake configuration for this project has 2 basic artifacts, `libevoengine.so` and the executable `evo` that is linked with it. Building the initial project follows the standard cmake method:

1. `cd build`
2. `cmake ..`
3. `make`

To run the program, while still in `build/` just execute `./evo`.

Initial project directory structure

```
├──── build/
├──── engine/
│    ├──── include
│    │    ├──── evo.h
│    │    └──── logger.h
│    ├──── src/
│    │    └──── logger.cpp
│    └──── CMakeLists.txt
├──── CMakeLists.txt
├──── app.cpp
└──── README.asciidoc
```

In this program, `app.cpp` is our `main` entrypoint and it includes `engine/include/evo.h` that acts as a single point of inclusion for all engine header files. Right now, `app.cpp` just outputs some text and waits for the user to press the enter key to exit. Output is sent to the console using the `Evo::Logger` class defined in `engine/include/logger.h`.

We'll use this structure as a starting point for building out the engine design.

Entrypoint

We're going to start by taking the entrypoint away from our client applications and move it into the engine library itself.

```
.
├──── build/
├──── engine/
│     ├──── include/
│     │     ├──── entrypoint.h
│     │     ├──── evo.h
│     │     └──── logger.h
│     ├──── src/
│     │     └──── logger.cpp
│     └──── CMakeLists.txt
├──── CMakeLists.txt
├──── app.cpp
└──── README.asciidoc
```

app.cpp

```
#include <evo.h>
```

engine/include/entrypoint.h

```
#include <iostream>

// We take control of program entry to ensure proper initialization
int main() {
  Evo::Logger l("myLogger");
  l.info("Press [ENTER] to exit...");
  std::cin.get();
}
```

This makes `app.cpp` pretty damn boring. However, we're going do something neat with that in a minute.

Why take main

We want to have control over program initialization, parameter parsing, and early validation. This all requires having access to the point at which the program starts. Sure the client application could define a main method and include our headers, but the only way to ensure proper engine startup is by taking control of it and treating the client application as more of an "application definition" instead of actual C++ application. To do this, let's introduce a way for our client to hook into the engine and (in the future) extend it.

Abstracting an application

engine/include/application.h

```
#pragma once

namespace Evo {
  class Application {
    public:
      Application();
      virtual ~Application();
  };

  // To be defined by the client application
  Evo::Application* createApplication();
}
```

app.h

```
#include <evo.h>

class App : public Evo::Application {
  public:
    App();
    ~App();

  private:
    std::unique_ptr<Evo::Logger> logger;
};
```

app.cpp

```
#include "app.h"

App::App() {
  logger = std::make_unique<Evo::Logger>("App");
  logger->info("App initialized");
}

App::~App() {
  logger->info("App destroyed");
}

Evo::Application* Evo::createApplication() {
  return new App();
}
```

engine/include/entrypoint.h

```
#include <iostream>

// We take control of program entry to ensure proper initialization
int main() {
  Evo::Logger l("myLogger");

  auto app = Evo::createApplication();

  l.info("Press [ENTER] to exit...");
  std::cin.get();

  delete app;
}
```