CEX – The Standard v1.0.0

Motive

C++ is a powerful tool. Some may argue too powerful. Constraints in a setting of abundance can prevent mind-splintering and inconsistent paradigms.

CEX (or C-Extended) aims to be a C++ subset *in between* C and C++ which takes advantage of existing C++ compilers, community and libraries while providing enough stylistic/implementation restrictions for an "idiomatic approach" to be meaningful.

Caveats

- CEX is non-OOP as is conventionally understood (no inheritance, polymorphism).
- Constructors/destructors are also not used in the conventional sense (see `new`).
- No 'class' keyword (yes structs are technically the same – more on this later).
- No getter/setter mentality. Struct members are accessed directly.

So – what *can* you do?

Let's start with a simple example.

```
struct Vec3D {
        float x, y, z;
}
```

Here we have a struct of a 3D vector.

Normally, we may be tempted to do something like this:

```
class Vec3D {
private:
        float x, y, z;
```

```
public:
        Vec3D(float x, float y, float z) { ... }
        ...
        void setX(float x) { ... }
        void getX() { ... }
        ...
        void normalize() { ... }
}
```

And so on.

There are many implications baked into the way this has been setup. Namely:

- This class structure encourages initialization using the `Vec3D(...)` constructor.
- It clearly discourages *direct* access to the `x, y, z` members.
- Some useful self-referential mutations (like `normalize()`) seem available.

How would this be done according to CEX?

```
struct Vec3D {
        float x, y, z;
        void normalize(void);
}
```

To highlight the difference, the developer is now in a position to initialize, alter and define the data structure through built-in and C-like struct manipulations.

What remains the most intact is the self-referential `normalize()` method which requires no special OOP-related features other than being self-referential (function pointers in structs are possible in C).

Using `Vec3D`

To continue the previous example, a use case is provided below:

```
struct Vec3D {
        float x, y, z;
        void normalize(void);
}
```

```
int main(void) {

        Vec3D pos1, pos2;

        pos1.x = 0;
        pos1.y = 0;
        pos1.z = 0;

        pos1.x += 3.4f;
        pos1.y -= 9.8f;
        pos1.normalize();

        pos2 = pos1;

        return 0;
}
```

Not bad. To the eyes of a C programmer who wants a little more, this is glorious but still missing a few things.

Introducing `make` and `copy`

Let's please not talk about Rust and pretend that these keywords are completely new.

- `make` ought to replace the constructor model and instead return an initialized copy.
- `copy` ought to return an identical copy of the struct.

```
struct Vec3D {
        float x, y, z;

        Vec3D make(void);
        Vec3D copy(void);

        void normalize(void);
}
```

And so:

```
int main(void) {

        Vec3D pos1 = Vec3D::new();
        // Do stuff to pos1...
        Vec3D pos2 = pos1.copy();
}
```

This saves us from the annoying labor of manually initializing the struct every time we'd like to use it, and also provides the means to "construct" it using function args.

Warning: Any avid C++ dev paying attention would be seething to have noticed that `Vec3D::new()` was called without an associated object.

This is because it needs to be declared as a static-member function.

In reality:

```
struct Vec3D {
        float x, y, z;

        static Vec3D make(void);
        Vec3D copy(void);

        void normalize(void);
}
```

Nice.

The `make` method

A closer look at the make method would reveal that it must be possible to return an initialized copy of the struct without referring to an existing object (hence the entry about the static declaration).

In `Vec3D` case, initializing all of the members to `0` was done manually in the previous entry:

```
Vec3D Vec3D::make() {
        return (struct Vec3D) {
                .x = 0,
                .y = 0,
                .z = 0,
        };
}
```

Clean and concise, we make use of struct value-initialization syntax inline with the return statement to produce the desired copy.

The `copy` method

This syntax for this method in C++ is actually quite nice. As a copy of the dereferenced `this` pointer can be returned as:

```
Vec3D Vec3D::copy() {
        return *this;
}
```

If this isn't programmer art I don't know what is.

Adding the `abs` method

It would be inconvenient to develop the `normalize()` method without having easy access to the `Vec3D`'s length in cartesian space.

```
struct Vec3D {
        float x, y, z;

        static Vec3D make(void);
        Vec3D copy(void);

        float abs(void);
        void normalize(void);
}
```

Where,

```
float Vec3D::abs() {
        float x, y, z;
        x = this->x;
        y = this->y;
        z = this->z;

        return sqrt(x*x + y*y + z*z);
}
```

Note: for those concerned about performance reduction when "copying" `x, y, z` out of their `this->*` counterparts, I'd recommend you check out godbolt.

Wrapping things up

Completing the previous example with function definitions:

```
#include <cmath>

struct Vec3D {
        float x, y, z;

        static Vec3D make(void);
        Vec3D copy(void);

        float abs(void);
        void normalize(void);
}

Vec3D Vec3D::make() {
        return (struct Vec3D) {
                .x = 0,
                .y = 0,
                .z = 0,
        };
}

Vec3D Vec3D::copy() {
        return *this;
}
```

```
float Vec3D::abs() {
        float x, y, z;
        x = this->x;
        y = this->y;
        z = this->z;

        return sqrt(x*x + y*y + z*z);
}

void Vec3D::normalize() {
        this->x = this->x / abs();
        this->y = this->y / abs();
        this->z = this->z / abs();
}

int main(void) {
        Vec3D pos1, pos2;

        pos1 = Vec3D::make();

        pos1.x += 3.4f;
        pos1.y -= 9.8f;
        pos1.normalize();

        pos2 = pos1.copy();

        return 0;
}
```