CEX – The Standard v2.3.3

Amendments since the previous version (v2.1)

1. Polymorphism was re-introduced (v2.1.0):

- To support the `Type::from(...)` feature.
- This would also support multiple ways to `Type::make(...)` a struct.

Amendments to the previous version (v2.2)

1. Introduction of templating with caveats (v2.2.0):

- Introduced **Explicit template specialization** as an allowed feature.
- This comes with the caveat of NO header function definitions.
- Banned **Implicit templates** and **Header function definitions**.
- The motive for this is simple: terrible compile times and the blasphemy of writing source code in headers.
- Added the `free` special method.
- Exists as the associated method equivalent of the destructor.
- Some structs allocate / open / initialize things which need to be deallocated / closed / uninitialized.

2. Tighter usage of `using` directives and `namespace` aliasing (v2.2.1):

- To avoid polluting header files, `using` directives must be used inside of source files (never header files).
- Other standards also agree that `using namespace` should *never* be used.
- Namespace aliasing (by `namespace X = std::whatever`) is OK in source files.

Amendments to the previous version (v2.3)

1. Introduced guidelines for compile flags during development (v2.3.0):

- Half of the battle is getting the compiler on your side – these are just starter guidelines.

2. Standard updating automation introduced (v2.3.1).
3. Formatting fix (error in the markdown) (v2.3.2).
4. Added `copy` as a special function (v2.3.3).

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.

Values

- Explicitness,
- Consistency,
- Transparency,
- Human readability,
- Bottom-up design,

Caveats

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

Permitted Features

- Structs,
- Namespaces,

- **Polymorphism** (v2.1.0),
- Methods (member/static),
- **Explicit template specialization (v2.2.0),**
- Operator overloads (`<<` , `==` , `>>` , etc),
- Struct member and associated (static) functions,

Disallowed Features/Keywords

- Classes,
- Exceptions,
- Getters/setters,
- Virtual methods,
- Default arguments,
- Constructors/Destructors,
- **Implicit templates (v2.2.0)**
- **Header function definitions (v2.2.0)**
- ~~Polymorphism (one name, one function)~~ (v2.1.0),

Recommended Alternatives

- Structs,
- Return values,
- Struct member access,
- Composition and templates,
- See `std::optional` ,
- Use `make` / `free` ,
- Use **Explicit template specialization**,
- Use **Explicit template specialization** with source function definitions,

Adapting to a C++ World

It must be said that many of CEX's idioms are fundamentally incompatible with much of the existing C++ codebases out in the world. This must be reconciled by:

1. CEX being a personal/organizational choice,

2. By design, working within the C++ ecosystem unlocks vast resources,
3. All other C++ libraries/codebases *will work* in CEX projects if included as intended,
4. When preferred or absolutely necessary, incompatible features/implementations can be wrapped within a CEX compliant API.

Compile Flags

Half of the battle is configuring the compiler to be on your side.

The following compiler/linker flags are *required* for development. I don't really care what you do for production:

```
CXXFLAGS=-std=c++20 -Wall -pedantic -Wextra -Werror
LDFALGS=-fsanitize=address
```

Note on previous C++ editions: Special projects may have special use cases. If you know what you're doing this is fine, but it may require some creative flexibility to adhere to the standard (especially parts which may rely on newer language features).

Though there aren't many, it is something to consider. If it doesn't matter to you, use the newest one by default.

Naming

Special Methods

There are a list of reserved method names for structs which perform *generally* similar behaviours which can be intuitively reused across different types.

Namely, `make`, `from`, `to`, `preset`, `copy` and `free`.

- `make` – reserved for the generation of a blank or empty struct (NOT for default values – see `default`),
- `from` – reserved for type conversions *from* another type,
- `to` – reserved for type conversions *to* another type (usually of an external or standard library),

- `preset` – similar to `make` but initializes the struct members to a set of default values rather than 0-values (or the equivalent thereof).
- `free` – the opposite of `make` – if anything was initialized / dynamically allocated / opened and needs to be de-initialized / deallocated / closed, this is the place.

As of (v2.1.0), polymorphism is supported. As a result, structs can be made `from` a variety of different types and by extension made *into* a bunch as well.

Rationale:

Examples

```
struct Fahrenheit {
        float temp;

        static Fahrenheit make(void);
        static Fahrenheit from(Celcius& t);
        static Fahrenheit preset(void);

        // Primary template for `to()`
        template<class T>
        T to(void);

        // Explicit specialization for T=Celsius
        template<>
        Celsius to<Celsius>(void);
        ...
}

struct Celsius {
        float temp;

        static Celsius make(void);
        static Celsius from(Fahrenheit& t);
        static Celsius preset(void);

        // Primary template for `to()`
        template<class T>
        T to(void);
```

```
        // Explicit specialization for T=Fahrenheit
        template<>
        Fahrenheit to<Fahrenheit>(void);
        ...
}

Fahrenheit Fahrenheit::make() {
        return Fahrenheit {
                .temp = 0,
        };
}

Fahrenheit Fahrenheit::from(Celsius& t) {
        return Fahrenheit {
                .temp = t.temp * 9 / 5 + 32,
        };
}

Fahrenheit Fahrenheit::preset(void) {
        return Fahrenheit {
                .temp = 212.0f,
        }
}

template<>
Celsius Fahrenheit::to<Celsius>() {
        return (this->temp - 32) * 5 / 9;
}

Celsius Celsius::make(void) {
        return Celsius {
                .temp = 0,
        };
}

Celsius Celsius::from(Fahrenheit& t) {
        return Celsius {
                .temp = (t.temp - 32) * 5 / 9,
        };
}
```

```
Celsius Celsius::preset() {
        return Celsius {
                .temp = 100.0,
        };
}

template<>
Fahrenheit Celsius::to<Fahrenheit>() {
        return (this->temp * 9 / 5) + 32;
}
```

File extensions

All CEX projects must use the `.cpp` file extension for source files and the `.hpp` file extension for header files.

Rationale: Although the `.c++` and `.C` extensions are disqualified by default due to special characters or filesystem indifference to capitalization, `.cxx` and `.cc` were considered to be less popular than `.cpp`. In the spirit of setting a consistent standard, the most common extension was the chosen one.

Namespaces

All namespaces must be named with lower-case and underscore-separated names (no upper/lower camel case) and match the name of the header/source file is encapsulates.

Example:

```
// project.hpp
namespace project {
        void act() { ... }
}

// project.cpp
void project::act() { ... }
```

Rationale: Exposing code modules or source files to the global scope of symbols increases the risk of collisions to no benefit when done correctly.

Quick rules for namespaces

- Never `using namespace` anywhere, *ever*.
- All `using` directives must go in source files.
- Namespaces can be aliased (NOT `using namespace`) in source files (minimize).

```
// project.hpp
using std::whatever; // NO
using namespace std; // NEVER
...
namespace project {
        struct Project { ... };
}

// project.cpp
using std::whatever; // Yes
using namespace std; // STILL NO
...
using project::Project; // Yes
...
namespace fs = std::filesystem; // OK
```

Informal module system

Matching the namespace name with the header/source combo informally introduces the concept of modules in the build scheme. Although modules are officially being streamlined into newer C++ standards, it's hardly catching on or working

An example project structure could resemble the following:

```
inc
\_ project.hpp
\_ module1.hpp
```

```
src
\_ project.cpp
\_ module1.cpp
```

In which each header/source pair are implicitly glued as modules by the namespace convention.

Alternatively:

```
project
\_ inc
        \_ project.hpp
\_ src
        \_ project.cpp

module1
\_ inc
        \_ module1.hpp
\_ src
        \_ module1.cpp
```

In both cases, namespace and filesystem layouts are used to group code in logical modules as much as possible.

Multiple headers/sources in one module

In case multiple header/source files are wanted to share the same namespace, the module must instead be created under it's own directory (`project/`) in this case and contain a separate `src` and `inc` directory in which all of the sub-sources and sub-headers will be stored, respectively.

For example:

```
inc
\_ project.hpp
\_ module1.hpp
\_ module2
```

```
        \_ module2.hpp
        \_ inc
        \_ src
```

At which point the `module2.hpp` header would declare the module's matching namespace and allow sub-sources and headers to complete the implementation therein.

One could evaluate an expanded tree of this as:

```
inc
\_ project.hpp
\_ module1.hpp
\_ module2
        \_ module2.hpp
        \_ inc
        |       \_ submodule1.hpp
        |       \_ submodule2.hpp
        \_ src
                \_ submodule1.cpp
                \_ submodule2.cpp
```

Where,

```
// module2.hpp
namespace module2 {
        namespace submodule1;
        namespace submodule2;
}

// subheader1.hpp


// subsource1.cpp
```

Headers

Please see the *Names and Order of Includes* section of the *Google C++ Style Guide* https://google.github.io/styleguide/cppguide.html#Names_and_Order_of_Includes.

Structs

Remember there are no class declarations.

Structs must be named in the upper camel case format, namely prevent name collisions with parent namespaces which could have good reason to match (a `project` namespace containing a `Project` struct).

Example:

```
// project.hpp
namespace project {
        struct Project { ... }
}

// project.cpp
using project::Project;
```

Rationale: The default accessibility is public and structs are typically thought of passive data carriers. This encourages the model of associated functions and transparent data types as opposed to the tightly encapsulated getter/setter model typically associated to classes.

Inheritance vs Composition

Under long a chain of inheritance (entity, animal, mammal, dog, pitbull, etc.), any changes in the parent types have immediate back-propagating consequences throughout the codebase.

These changes need not be breaking – a monolithic type hierarchy will inevitably impose greater leverage on parent types such that minor changes have greater and greater impact on the rest of the code.

To make matters worse, standard library and otherwise large/complex objects are strongly discouraged from being inherited to expand functionality. Thus, alternatives

and workarounds are needed depending on the types involved (subject to developer preference).

Instead, lateral type conversions (`to` and `from`) mixed with composition/wrappers are encouraged to minimize (albeit not eliminate) monolithic type hierarchies within code.

Rationale: Prevents back-propagated changes to entire codebases and monolithic type dependency hierarchies. Also encourages lateral type conversions.

Using Directives

When there are no collisions with other dependencies, apply `using` directives to structs/objects by default.

Example:

```
// project.hpp
namespace project {
        struct Project {
                void act(void);
        }
}

// project.cpp
using project::Project;

void Project::act() { ... }
```

Long chains of scope (e.g.: `util::rf::Reader`) which may pose confusion or name collisions with another namespace (e.g.: `util::io::Reader`), it is considered acceptable to *alias* namespaces to the nearest heterogenous parent (NOT `using` the namespace, *ever*).

Example (needs to be redone – I don't advocate namespaces within namespaces):

```
// util.hpp
namespace util {
        namespace io {
                struct Reader { ... }
```

```
        }

        namespace rf {
                struct Reader { ... }
        }
}

// main.cpp
namespace io = util::io;
namespace rf = util::rf;

int main(void) {
        io::Reader a;
        rf::Reader b;
        ...
        return 0;
}
```