Core object model
EO / EFL++

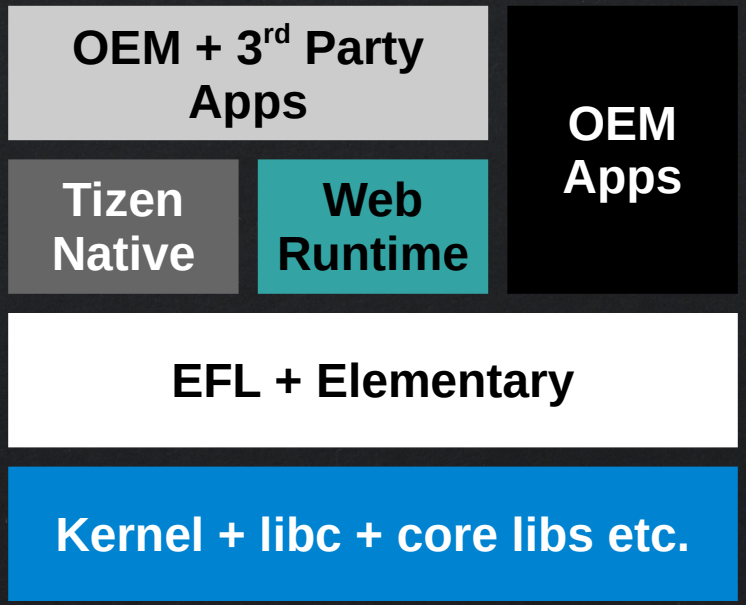
Carsten Haitzler
Samsung Electronics
Principal Engineer
Enlightenment/EFL Founder

c.haitzler@samsung.com

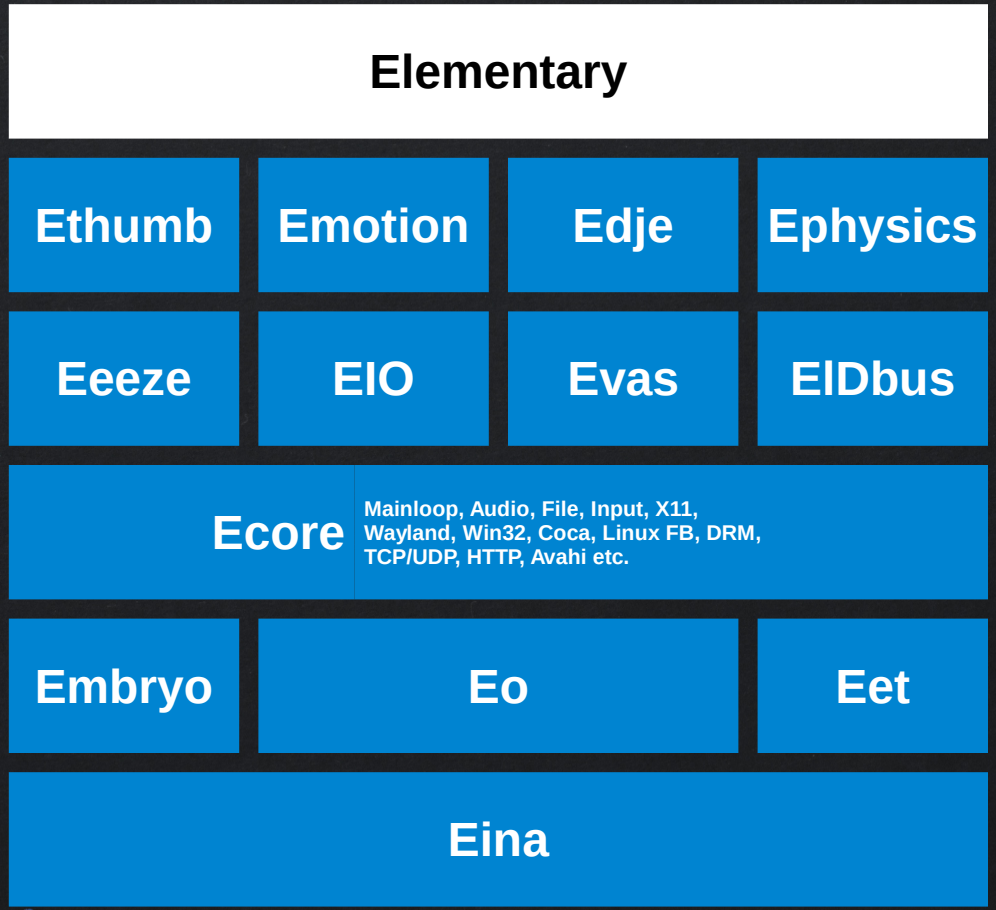
TIZEN™
DEVELOPER
CONFERENCE
2014
SAN FRANCISCO

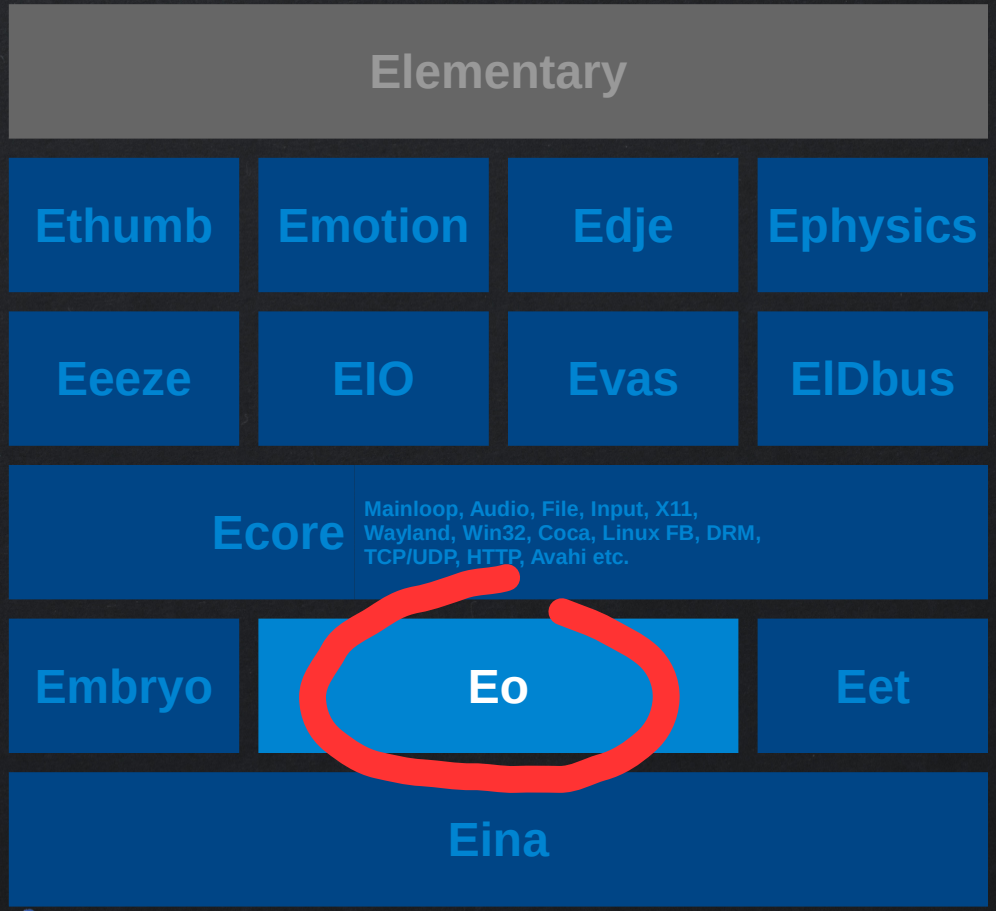
EFL + Elementary

- A toolkit somwhere between GTK+ and Qt in breadth and features
- Written in C
- Has a primitive object model of its own since its start
- Is at the core of Tizen today



* Really really really simplified diagram




Elementary
Ethumb
Emotion
Edje
Ephysics
Eeeze
EIO
Evas
EIDbus
Ecore
Mainloop, Audio, File, Input, X11,
Wayland, Win32, Coca, Linux FB, DRM,
TCP/UDP, HTTP, Avahi etc.
Embryo
Eo
Eet
Eina


EO

EO – our new base class

- **Before we had pseudo-objects**
  - Timers, Animators, etc.
  - Evas objects
  - Edje objects (inhrited from Evas)
  - Elementary objects (inherited from Evas and Edje)
  - And more...
- **EO unifies all of these with a single base class**
  - Done in C
  - Provides call safety and object access safety
- **EO provides binding generation for C++ … and soon LUA etc.**

EO features

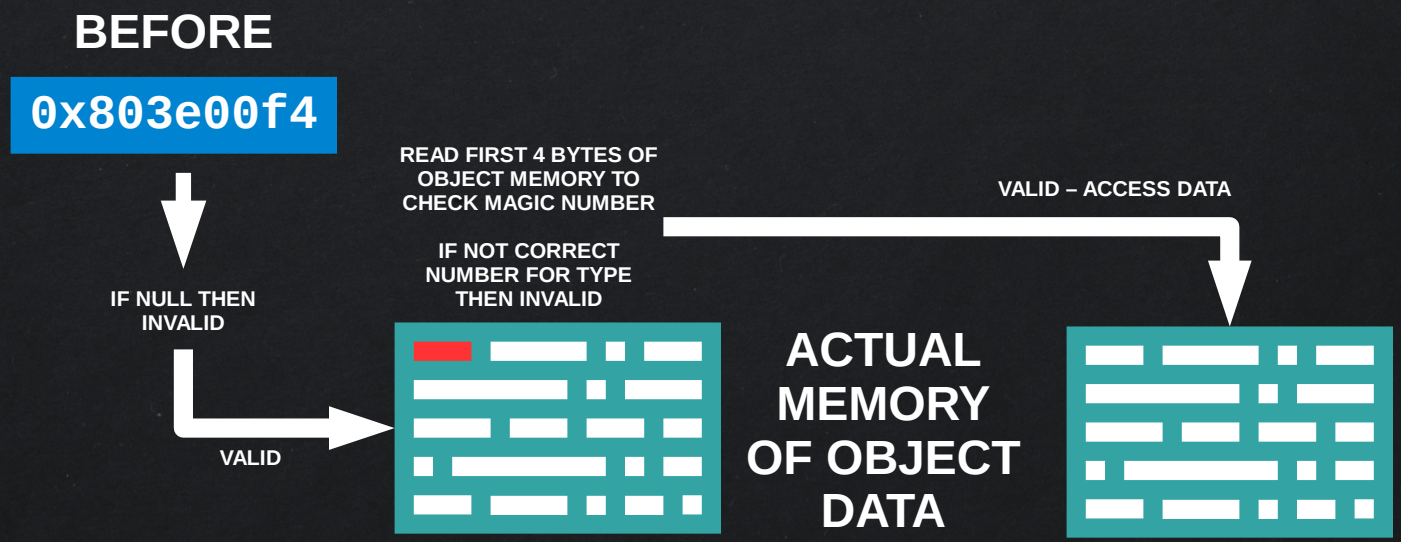
- **Single and multiple inheritance with overrides**
- **Plain interfaces**
- **Mixins**
- **Reference counting**
  - Weak references
  - Cross-references between objects
- **Event callbacks and control for all objects**
- **Parent + child tree (children auto deleted)**
- **Key + value attachment to all objects**
- **Runtime checks**
  - Invalid reference access checks
  - Method/class/type checks

EO – Why?

- **You just re-invented GObject!**
  - No – our base class is more extensive
  - Features built around unifying and providing compat for existing EFL
  - We now auto-generate the boilerplate code
  - We auto-generate legacy compatibility binding functions for C
  - We have runtime method checks, not compile-time
  - We have an elaborate object handle indirection scheme for safety

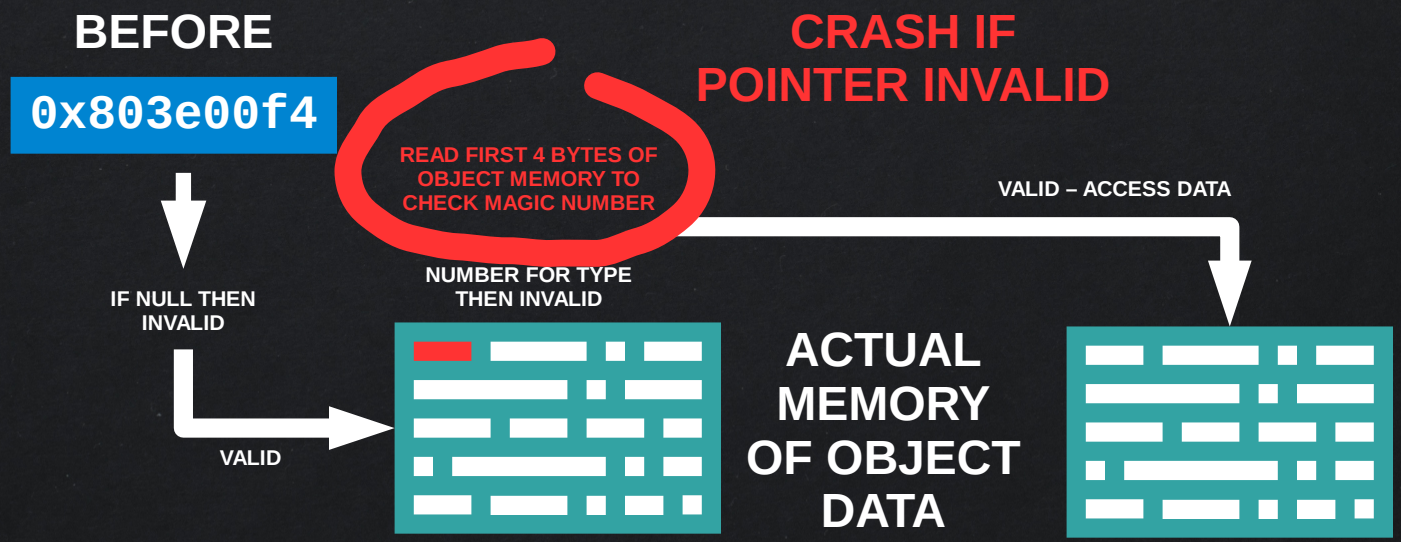
EO – Object reference safety?

- In C and C++ most objects are pointers (Qt, GTK+, EFL)
- We now hide pointers and use indirection



EO – Object reference safety?

- In C and C++ most objects are pointers (Qt, GTK+, EFL)
- We now hide pointers and use indirection

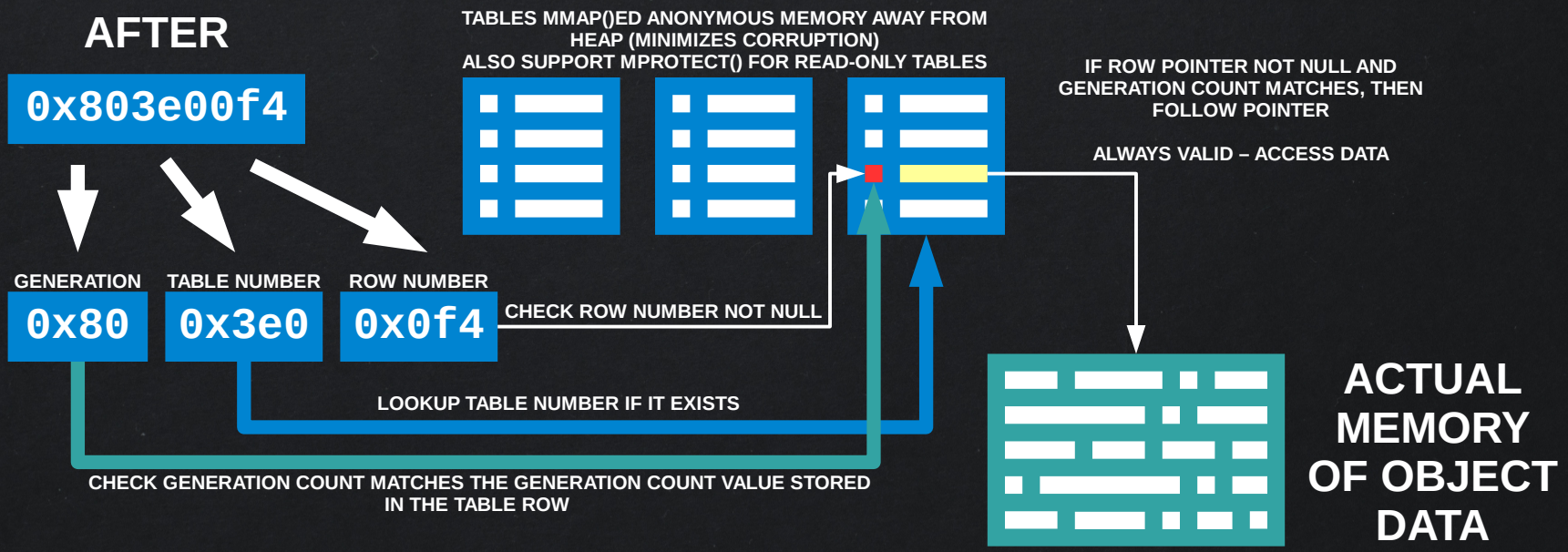


So developer uses invalid access – so what?

- **Bug reports always filed for an EFL bug**
  - Backtrace always ends inside EFL – thus “it must be an EFL bug”
  - EFL developers very often debugging applications, not EFL
    - Need to prove application is at fault – time consuming
  - Wastes EFL developer time
  - Means apps crash while a user is busy doing something important
  - Really annoying to keep explaining what backtraces say
- **Need a solution that is safer...**

EO – Object safety added in

- Object “pointers” are reference IDs

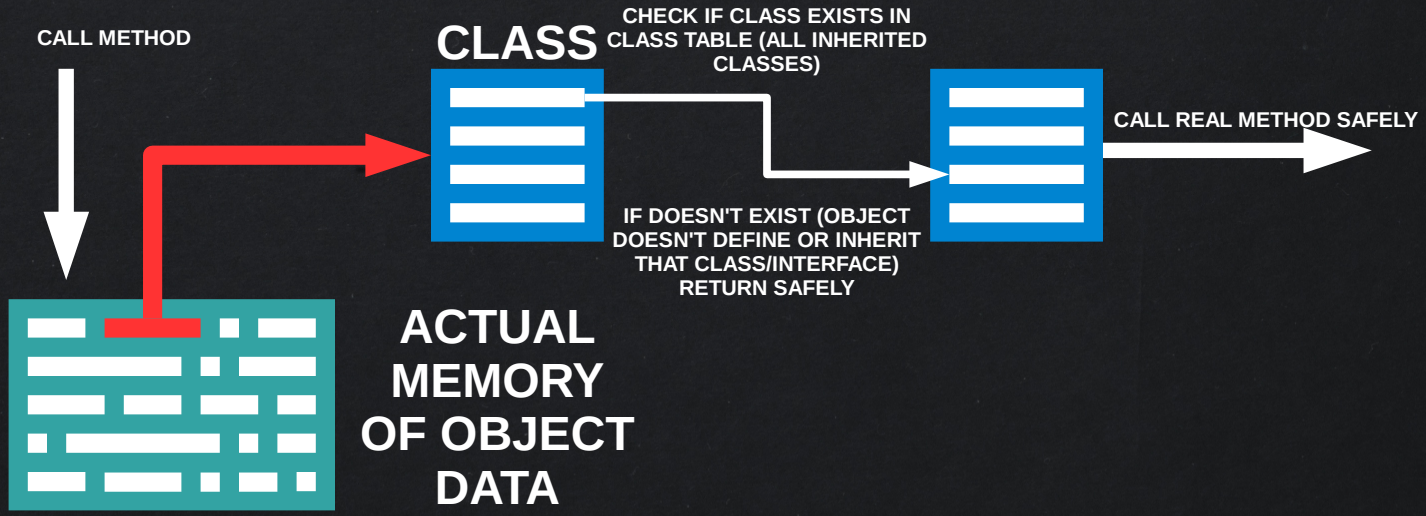


So a pointer is not a pointer?

- **Yes. Pointers only used for compatibility :(**
- **On 32bit, 9 bits are for Generation count, the rest for table + row**
  - One in 512 chance of a false positive on valid row
- **On 64bit, 29 bits for Generation count, the ret for table + row**
  - On in ~500 million chance of a false positive on a valid row
- **Even if a false positive sneaks through**
  - We found **A** valid object – maybe right, maybe wrong
  - If wrong object, type checks happen due to runtime method lookup
- **Worst case – you manipulate an unitended object – no crash**
  - No worse than before

Runtime dynamic method lookup?

- Yes. If method is invalid for the class – it is skipped
- All methods can be batched to save object lookup cost




So... C++ eh?

So what has this got to do with C++?

- **Just like C++...**
  - EFL now has constructors and destructors
  - EFL can inherit methods from parents
  - EFL can override methods that are inherited
  - EFL can multiply inherit and even do mixins directly
- **Also like Javascript, LUA etc.**
  - EFL objects are reference counted for auto-cleanup when all references go
  - EFL objects have properties as well as methods
  - EFL can just tag data on objects like simply adding values to a table by key

C & C++ style with EFL

C style “do” call – one method per call

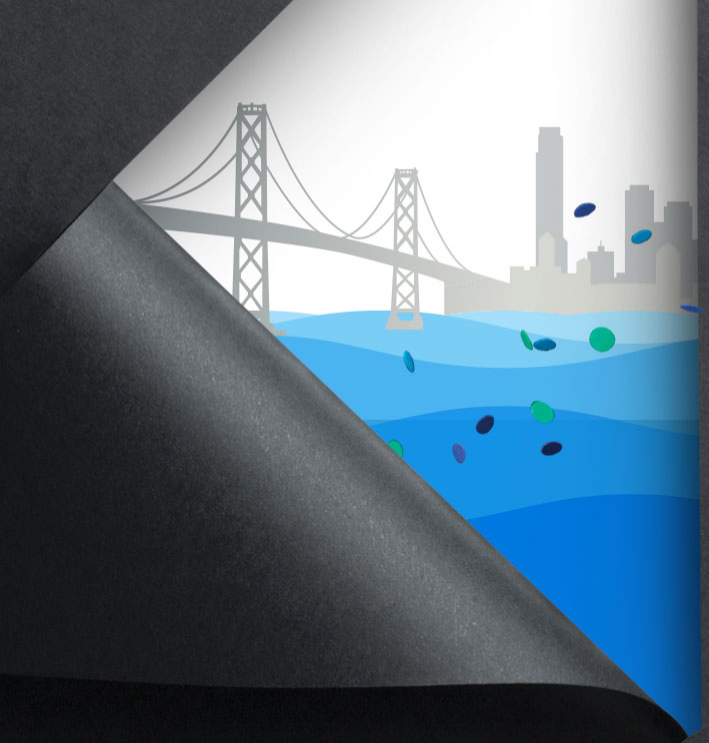
```
eo_do(obj, efl_text_style_set(style));
eo_do(obj, efl_text_set(text));
eo_do(obj, efl_gui_size_get(&width, &height));
```

C style batched calls – 3 methods per call

```
eo_do(obj,
      efl_text_style_set(style),
      efl_text_set(text),
      efl_gui_size_get(&width, &height));
```

C++ style object calls

```
obj->text_style_set(style);
obj->text_set(text);
obj->gui_size_get(&width, &height));
```



What this looks like

But EFL is C, not C++ ?????

- **We now write out class definition in “eo files”**
- **Eolian generates the boilerplate C + EO code to create a class etc.**
- **From this data Eolian generates C++ headers**
  - Calls match 1:1 from C classes/methods/properties to C++
  - These C++ classes can be inherited from etc. like normal C++
  - Since they are only headers only, the C++ ABI is in fact C, not C++
    - This avoids all the common C++ ABI issues
  - We have standardized on C++11 STL for base datatypes
    - Provided manual bindings between EFL Lists, Hashes etc. to STL ones

C++ :(

- **To be honest – EFL devs don't like C++**
- **We're never going to port EFL to C++**
  - Over or dead stinking corpses
- **BUT... we understand others like C++**
  - And a lot of them keep asking us, as we try our best to ignore them
  - And they get upset when they can't just “new” and “delete”
- **So we're willing to help and oblige (GASP!)**
  - As long as we don't have to move to C++
- **And we have to do little to no maintenance to keep the support**

No maintenance? ORLY?

- **Eolian C++ generates the C++ headers directly from .eo files**
  - Whenever we add classes or methods, they get added with a re-run
- **The same method will add LUA bindings**
  - Same classes, methods and properties as C/C++
  - Auto-generated just like C++
  - Provides an alternative to native
  - Acts as a test case for dynamic languages
    - Once proven and useful it can expand to Javascript (v8), Python and others
- **And yes – we're being optimistic**

Sample eo file

```
class Tst (Eo_Base)
{
   eo_prefix: tst;
   data: Tst_Data;
   properties {
      name {
         set { /*@ This sets the name of the tst object */
         }
         get { /*@ This gets the name of the tst object if set */
         }
         values {
            const char *name; /*@ The name of the tst object as a C string */
         }
      }
      size {
         set { /*@ Sets the size of the object, on success returns EINA_TRUE */
            return Eina_Bool; /* returns EINA_TRUE on success */
         }
         get { /*@ This gets the size set */
         }
         values {
            int size; /*@ The size in pixels */
         }
      }
   }
```

```
   methods {
      activate { /*@ This method will activate the tst object, and when
                  * called, any events listening to activated will be
                  * triggered */
         params {
            @in int number; /*@ The number of pixels to activate */
            @in const char *string; /*@ A label to display on activation */
         }
         return Eina_Bool; /* If activation succeeds, returns EINA_TRUE */
      }
      disable { /*@ This disables the tst object to the level intidicated */
         params {
            @in int level; /*@ This is the disabling level to use */
         }
      }
   }
   implements {
      Eo_Base::constructor;
      Eo_Base::destructor;
   }
   events {
      activated; /*@ When the tst object has been activated */
      disabled; /*@ When the tst object has been disabled */
   }
}
```

Using the class in C

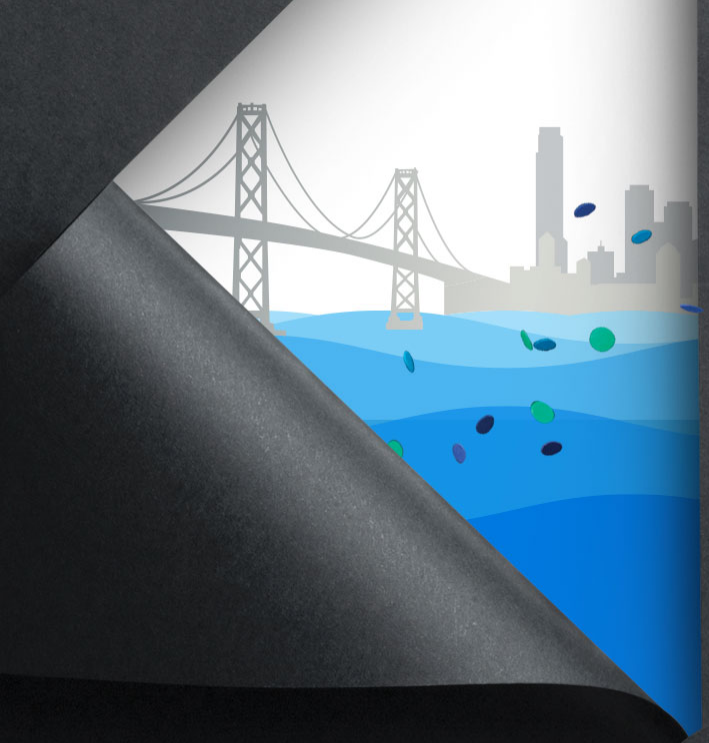
```
#include <Eo.h>
#include "tst.eo.h"

int main(int argc, char **argv) {
   eo_init(); // init eo
   Eo obj = eo_add(TST_CLASS, NULL); // create a new object of the TST class
   eo_do(obj,
         tst_name_set("Smelly"),
         tst_size_set(100));
   eo_do(obj, tst_activate(37, "Chickens"));
   eo_do(obj, tst_disable(99));
   eo_del(obj); // delete the created object
   return 0; // exit cleanly
}
```

Using the class in C++

```
#include <Eo.h>
#include "tst.eo.hh"

int main(int argc, char **argv) {
   efl::eo::eo_init init; // init eo
   tst *obj = new tst(NULL); // create a new object of the TST class
   obj->name_set("Smelly"),
   obj->size_set(100),
   obj->activate(37, "Chickens");
   obj->disable(99);
   delete obj; // delete the created object
   return 0; // exit cleanly
}
```



Why should yo care?

Why care or get excited?

- **Developers can choose C or C++**
  - And eventually LUA and maybe Javascript, Python etc.
  - Maintains the same API and behavior just with language syntax changed
  - Lets you choose what is easier for you
- **Provides for a C++ API with minimal ABI issues**
- **Helps you create software more easily**
- **Provides more safety for your Apps at runtime even with mistakes**
- **Provides for another object model for the C world**
- **Coming to Tizen … soon**
- **Makes everything more complex, and we love complexity :)**


TIZEN™
DEVELOPER
CONFERENCE
2014
SAN FRANCISCO


Q&A?
Flames?
Rants?