Overview

- JS
  - How to compile
  - How it works
  - How to use

- C++
  - How to use
  - Advanced stuff
Before it gets too boring
# How to compile
Compiling EFL

# First download it
$ git clone git@git.enlightenment.org:core/efl.git
$ cd efl

# Then configure for nodejs
$ ./autogen.sh --prefix=/path/to/efl \  
    --with-js=nodejs

# Now we compile, test and install
$ make && make check && make install
$ cd src/examples
$ NODE_PATH=/path/to/efl/lib: \  
    node box_js_example_02.js
The JavaScript binding is made against the library v8 from Google. This is a well-known interpreter for JavaScript and is used by node.js.

libv8 is used by node.js, so javascript code can be ported to node.js and native bindings can be ported too.

node.js also uses libuv for loop event, while EFL uses ecore. Ecore library can be integrated, as proven by the integration with glib, with other loop events.

There is three options of configuration for the JavaScript binding:

1. Node.Js
2. libv8
3. libv8 + libuv
Ecore
Edje
EFL
Evas

Eolian JS

class.js.cc

C++ compilation

efl.node

V8
efl = require(efl);

win = new efl.ui.Win_Standard(null);

btn = new efl.ui.Button(win);
btn.text = "Good-Bye, World!"
btn.size = {120, 30};
btn.position = {60, 15};
btn.visible = true;

function callback_function()
{
    console.log('clicked');
}

btn.on('clicked', callback_function);

win.size = {240, 60};
win.visible = {true};
```javascript
const efl = require('efl');

const btn = new efl.ui.Button(win);
btn.on('clicked', callback_function);
```

```
libuv loop

Events (timers, fds, etc)

callback_function()

clicks, updates, etc
```
Node.Js

- Uses a library (libuv) for loop event
- Uses libv8 for JavaScript interpretation
- You can write native code to run in node.js if you load it through a require and renames its extension to node, e.g., efl.node
- Applications can be written completely in JavaScript and share the same event loop with EFL’s UI events
- It is the most straightforward way to write JavaScript standalone applications
You can configure the javascript binding to compile to libv8 directly, with or without libuv integration:

```
# To compile binding without libuv integration
$ ./configure --with-js=libv8
# To compile binding with libuv integration
$ ./configure --with-js=libuv
```

This allows native applications to embed JavaScript code directly by linking with libv8 and running a JavaScript code. If the application already uses ecore or glib loop, then it doesn’t need libuv integration and can create and control the UI from javascript while using the currently used loop event.
How it works
Node.JS

node process

```javascript
var efl = require('efl')

win = new efl.ui.Win(null, 'abc', 0);
win.title = 'title';
win.autohide = true;
```

uv_run() – nodejs event loop
Before demonstration
Show Twitter Example
C++

- Focus on performance
- Modern C++
- Use C++11 features
C++

- Focus on performance
- Modern C++
  - Use C++11 features
C++

- Focus on performance
- Modern C++ No QT API bullshit
- Use C++11 features
C++

- Focus on performance
- Modern C++ No QT API bullshit
- Use C++11 features
C++

- Focus on performance
- Modern C++ No QT API bullshit
- Use C++11 features Lambdas, yay!
Performance

- Thin wrapper over C \((\text{sizeof}(\text{Eo}*) \equiv \text{sizeof(}\text{wrapper}))\)
- All functions inlined
- All containers are thin wrappers over Eina containers
- No copy between C and C++ API
Performance

- Thin wrapper over C (sizeof(Eo*) == sizeof(wrapper))
- All functions inlined
  - All containers are thin wrappers over Eina containers
  - No copy between C and C++ API
Performance

- Thin wrapper over C ($\text{sizeof(Eo*)} == \text{sizeof(wrapper)}$)
- All functions inlined
- All containers are thin wrappers over Eina containers
- No copy between C and C++ API
Performance

- Thin wrapper over C (sizeof(Eo*) == sizeof(wrapper))
- All functions inlined
- All containers are thin wrappers over Eina containers
- No copy between C and C++ API
WTF is Modern C++?

High abstraction without sacrificing performance (goal is to have *better* performance)
Modern C++

- WTF is Modern C++?
- High abstraction without sacrificing performance (goal is to have better performance)
/// @brief Eo Constructor.
///
/// Constructs the object from an Eo* pointer stealing its ownership.
///
/// @param eo The Eo object pointer.
///
explicit bar(Eo* eo) :
    efl::eo::base(eo)
{}
Thin wrappers

```
efl::eina::range_list<evas::object>
foo() const
{
    Eina_List * _tmp_ret;

    _tmp_ret = ::bar_foo(_eo_ptr());

    return efl::eolian::to_cxx
        <efl::eina::range_list<evas::object>>(_tmp_ret,
            std::tuple<std::false_type,
                std::false_type>());
}
```
Modern C++

```cpp
list = a::foo();

for (auto&& c : list)
{
    c.visible_set(false);
}
```

```cpp
for (auto& c : list)
{
    if (c::eina::range::list::evas::object::list = a::foo());
```
Before type traits
template <typename T>
class inarray
  : public eina::if_<
    <eina::is_pod<T>, _pod_inarray<T>,
    _nonpod_inarray<T> >::type
{

template<typename ContiguousIterator>
stringshare(ContiguousIterator i,
    ContiguousIterator j,
typename eina::enable_if
    <eina::is_contiguous_iterator
    <ContiguousIterator>::value>::type* = 0)
: _string(::eina_stringshare_add_length
    (&*i, j - i))
{
template <typename T, typename Enable = void>
struct is_contiguous_iterator
 : indirect_is_contiguous_iterator<T>
{
};

template <>
struct is_contiguous_iterator
 <std::string::const_iterator> : true_type
{
};

template <>
struct is_contiguous_iterator
 <std::string::iterator> : true_type
{
};

...
std::string s = "somestring";
stringshare a (s.begin(), s.end());
Modern C++ is all about typing

Types can help us with performance because they convey static information

Type Traits (which are, again, types) allow us to use these static information for optimizations
Show C++ Example
struct ColourableCircle :
    efl::eo::inherit<ColourableCircle, ::ns::Colourable>
{
    ColourableCircle(int rgb) :
        inherit_base(::ns::Colourable
            ::rgb_24bits_constructor(rgb))
    {}

    int colour_get()
    {
        return 0xFF0000;
    }
};
```cpp
void foo(::ns::Colourable c) {
    std::cout << c.colour_get() << std::endl;
}

ColourableCircle c(0x0);
foo(c);
```
Thank You

Q&A