Foreign Relations

The State of C++ Interop in D

Monday 2022-08-01 - London - Mathias 'Geod24' LANG

https://geod24.github.io/DConf2022/

안녕하세요

- Examples from **BPFK** (🇰🇷)
- Views are my own, not my employer, present or past

BPFK: Why D?

- Strongly typed system programming language
- Prototype to production in no time
- **Best C++ integration**, great C integration

Prototyping Agora

- C++ code
- Go implementation

Supported features

- Pretty much everything
- `class` / `struct` , `ref` , pointers, `const` , `nothrow` …
- Templates (!)
- Operator overloads (!!)
- Exceptions (!!!)

Step 0: Organization

```
+ agora
|- dub.json
|- source/agora
|- source/scpd
|- source/scpp
```

Step 1: Build system

```
    "preGenerateCommands": [
        "$DUB --verbose --single scripts/build_scpp.d"
    ],
    "sourceFiles-posix": [
        "source/scpp/build/*.o"
    ],
    "sourceFiles-windows": [
        "source/scpp/build/*.obj"
    ],

    "versions": [ "_GLIBCXX_USE_CXX98_ABI" ],
    "dflags": [ "-extern-std=c++17" ],
    "lflags-posix": [ "-lstdc++" ],
```

Step 2: Know your target

- `CppRuntime_Clang` => OSX, Linux, Windows
- `CppRuntime_Gcc` => Linux (OSX in the future?)
- `CppRuntime_Microsoft` => Windows

Step 3: The simple stuff

```d
extern(C++) struct Foo { int a; }
extern(C++) void func1 (ref const(Foo) f);

extern(C++) void func2 (const(Foo*) f);

extern(C++) void func3 (const(Foo**) f);
```

```cpp
struct Foo { int a; };
void func1 (Foo const& f);

void func2 (Foo const* f);
// void func2 (Foo const* const f);

void func3 (Foo const* const* f);
```

It is D code: Follow D rules

Namespaces

```
extern(C++, "dlang", "awesome", "app") void awesomeFunc ();
// Don't do this:
extern(C++, dlang.awesome.app) void lessAwesome ();
static assert(        lessAwesome.mangleof ==
    dlang.awesome.app.lessAwesome.mangleof);
```

Smarter namespaces

```d
version (CppRuntime_Clang)
    enum StdNamespace = AliasSeq!("std", "__1");
else
    enum StdNamespace = "std";

// Mind the parenthesis!
public extern(C++, (StdNamespace)) struct equal_to (T = void) {}
```

KISS

```
public extern(C++, (StdNamespace)) struct pair (T1, T2)
{
    T1 first;
    T2 second;
}
```

- Mangling
- Vtable / Offset
- Size
- Lifetime functions (ctor/dtor/copy/move)

KISSS

- Mangling => `pragma(mangle, str)`
- Vtable / Offset => Tests
- Size => Tests
- Lifetime functions => `ref` / pointers / wrappers

Testing size

```d
static foreach (Type; GlueTypes)
    extern(C++) ulong cppSizeOf (ref Type);

/// size checks
unittest
{
    foreach (Type; GlueTypes)
    {
        Type object = Type.init;
        assert(Type.sizeof == cppSizeOf(object),
            format("Type '%s' size mismatch: %s (D) != %s (C++)",
                Type.stringof, Type.sizeof, cppSizeOf(object)));
    }
}
```

Testing layout

```
/// Contains the size and offset of a field within a struct
extern(C++) struct FieldInfo { long size, offset; }

static foreach (Type; GlueTypes)
    extern(C++) FieldInfo cppFieldInfo (ref Type, const(char)*);
```

```d
/// size & layout checks for C++ structs / objects
unittest
{
    foreach (Type; TypesWithLayout)
    foreach (idx, field; Type.init.tupleof) {
        auto object = Type.init;
        auto field_info = cppFieldInfo(object,
            Type.tupleof[idx].stringof.toStringz);

        assert(typeof(field).sizeof == field_info.size,
                    format("Field '%s' of '%s' size mismatch: %s (D) != %s (C++)",
                Type.tupleof[idx].stringof, Type.stringof,
                typeof(field).sizeof, field_info.size));

        assert(Type.tupleof[idx].offsetof == field_info.offset,
            format("Field '%s' of '%s' offset mismatch: %s (D) != %s (C++)",
                Type.tupleof[idx].stringof, Type.stringof,
                Type.tupleof[idx].offsetof, field_info.offset));
    }
}
```

std::map

```
#include <map>

template<typename K, typename V>
class Map {
    static Map<K,V>* make ()     { return new Map<K,V>(); }

    V& operator[] (K const& key) { return this->map[key]; }

    void insertOrAssign(const K& key, const V& value) {
        this->map.insert_or_assign(key, value);
    }

    std::map<K, V> map;
};

// Explicit instantiation
template struct Map<char const*, int>;
```

std::map on the other side

```
extern(C++, class):
struct Map (Key, Value) {
    extern(D) void opIndexAssign (Value value, const Key key)
    {
        this.insertOrAssign(key, value);
    }

    static Map* make ();
    ref Value opIndex (ref const Key key);
    private void insertOrAssign(const ref Key, const ref Value);
}
```

How it should be

```
#include <map>
template class std::map<char const*, int>;
```

```
import core.stdcpp.map;
alias MyMap = map!(const(char)*, int);
```

`std` bindings

- Currently in `core.stdcpp`
- Will be moved to another library
- `allocator`, `array`, `vector`, `string`, `exception`, `memory`, `string_view`, etc…
- But not `map`

C++ wrapper code

- You need it (template instantiation)
- Your most powerful ally
- Pass by `ref`
- Weird C++ code: wrap `throw`, return value, etc...

Gradual C++ replacement

- Replacing single functions is easy
- Replacing methods is also trivial
- Easy way to weed-out dependency / improve code quality

Features for C++ integration

- `extern(C++, [class|struct])`
- `extern(C++, ident|expression)`
- `core.attributes : gnuAbiTag`
- `pragma(mangle, str_or_decl, [str])`
- Copy constructor / interior pointers (`std::string`)
- DWARF exception handling
- `__traits(getTargetInfo, "something")`

What worked for us

- Good C++ code
- Extra C++ code from the start
- Explicit template instantiation or wrapper
- Pass by `ref` / pointer
- Hand-crafted and basic UT
- `-preview=in` (`const T&`)
- Not using DMD