Interfacing D With C++

by Walter Bright

dlang.org

C is the Lingua Franca

- Most every language has some sort of interface with C
- And, of course, the classic being C++ is built on top of C

C Interop

```
extern ( C ) {
    void* malloc(size_t);
    void free(void*);
}
```

C++ Interop?

- Name mangling
- Templates
- SFINAE
- Namespaces
- Overloading
- Argument Dependent Lookup

Inconceivable!

--The Princess Bride

- RTTI
- Virtual functions
- Exceptions
- Special member functions
- Operator overloading
- Const

Oh My!

Imposserous!

– The Wizard of Oz

You'd have to build a whole C++ front end into the language!

Or Maybe Not...

Don't have to *compile* C++, just have to *link* to it

D doesn't have an analog of everything C++ has, so if we can be a bit plastic on both sides...

```
extern (C++)
{
    uint foo(ref char* p);
}
```

Should connect to:

```
extern “C++”
{
    unsigned foo(char*& p);
}
```

D	C++
char	char
byte	signed char
ubyte	unsigned char
short	short
ushort	unsigned short
int	int
uint	unsigned
long	long long
ulong	unsigned long long

What About

```
extern “C++” void foo(long x);
```

(long doesn't seem to have a D analog)

```
struct __c_long {
    this(int x) { lng = x; }
    int lng;
    alias lng this;
}
```

Unsolved Const Problem

```
int ****const*** func();

?func@@YAPAPAPBQAPAPAPAPAHXZ

const(int ****)*** func();

?func@@YAPAPAPBQBQBQBHXZ
```

Struct Layout Matches C++

C++:

```
struct s { unsigned a; char c; double d; };
```

D:

```
struct s { uint a; char c; double d; }
```

Static members too!

Struct Member Functions

The same

Polymorphism (virtual functions)

- D classes have virtual functions
  - But object layout is different
  - `vtbl[]` layout is different

D Supports COM Interfaces

```
import std.c.windows.com;

interface IHello : IUnknown {
    extern (Windows) int Print();
}

class CHello : ComObject, IHello {
    HRESULT Print() {
        MessageBoxA(null, “hello”, null, MB_OK);
    }
}
```

Or Simply

```
extern (C++) class C {
    void func() { … }
}
```

Multiple Inheritance

Lord of the Rings

Not even once!

Floor Wax or Dessert Topping?

Value or reference type?

C++ Namespaces

```
namespace N {
    namespace M {
        void foo();
    }
}

namespace N {  // not closed
    void bar();
}
```

D Name Spaces

- module
- struct
- class
- mixin template

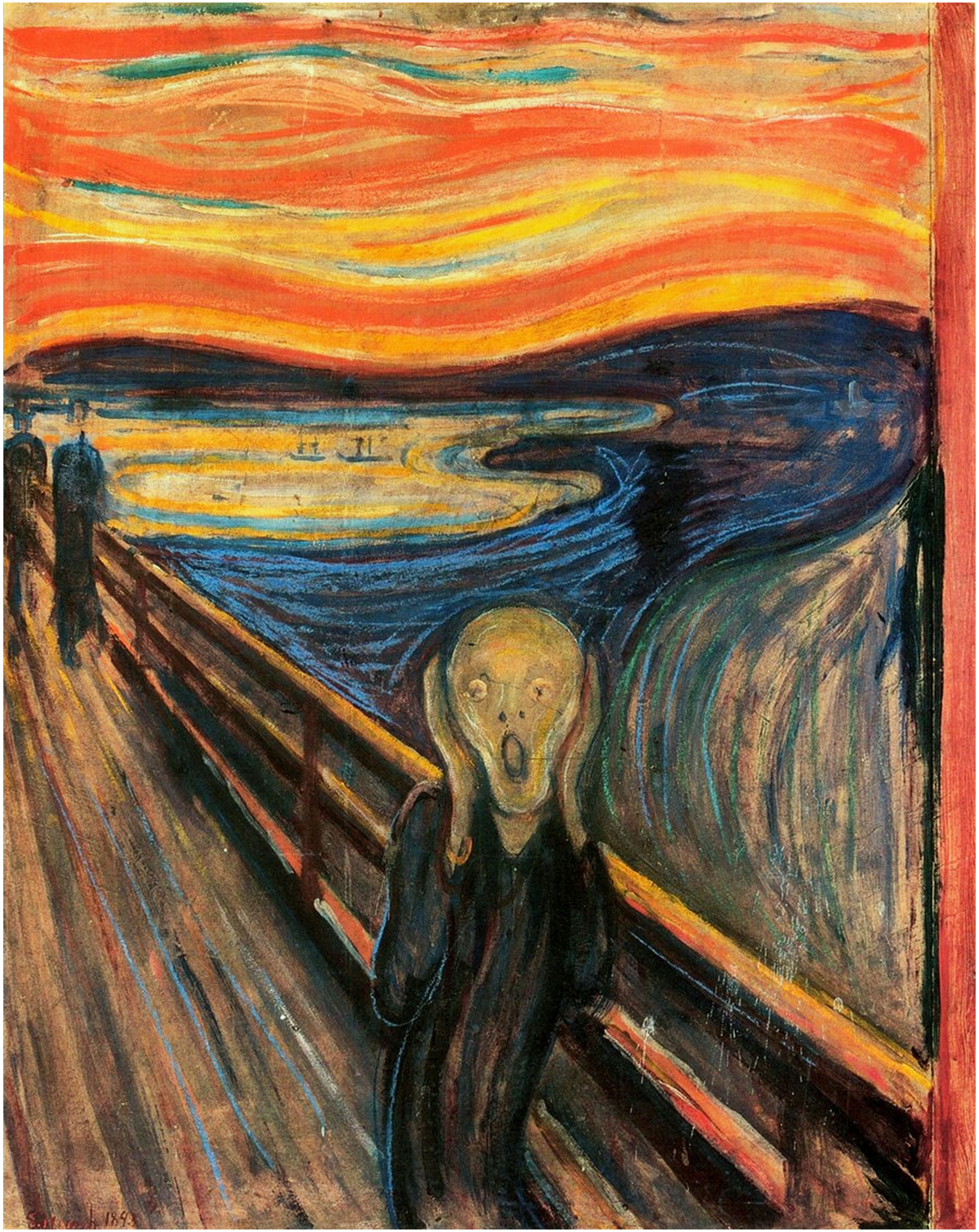
Extend C++ Declaration

```
extern (C++, N.M) {
    void foo();
}

extern (C++, N) {
    void bar();
}
```

C++ Templates

- SFINAE
- Partial ordering
- Dependent lookup
- Point of instantiation
- Primary template
- Template templates


E. Munch 1893

Ignore All That

It's just a name mangling problem.

C++:

```
template<class X, int C>
struct Boo {
    X v[C];
};
```

D:

```
extern (C++)
struct Boo(X, int C) {
    X[C] v;
}
```

Toto Too!

Wizard of Oz

C++:

```
template<class T>
   T func(T t) { return t; }

func(3);

??$func@H@@YAHH@Z
```

D:

```
extern(C++)
T func(T)(T t) { return t; }

func(3);

??$func@H@@YAHH@Z
```

Now It's Time to Justify My Existence

Interface to STL!

Let's try and hook up to

`std::vector<T>`

```
std.vector!int p;
func(p);
```

calls:

```
void func(std::vector<int, std::allocator<int> > *p);
```

```
extern (C++, std) {
    class vector(T, A = allocator!T) {
        final void push_back(ref const T);
    }
}
```

```
extern (C++, std) {
  struct allocator(T) {
    alias size_type = size_t;
    void deallocate(T* p, size_type sz) {
      (cast(__gnu_cxx.new_allocator!T*)&this).deallocate(p, sz);
    }
  }
}

extern (C++, __gnu_cxx) {
  struct new_allocator(T) {
    alias size_type = size_t;
    void deallocator(T*, size_type);
  }
}
```

Biggest Remaining Problem

- Catching C++ exceptions
  - which are by value
  - D exceptions are by reference

Tl,Dr;

- Can get pretty far
- Need to be flexible on both ends
- Interfaces to STL are not portable
- and requires non-trivial expertise

- It'll never be 100%
- But it's tractable
- And infinitely better than C wrappers
- No longer locked in to existing C++ code