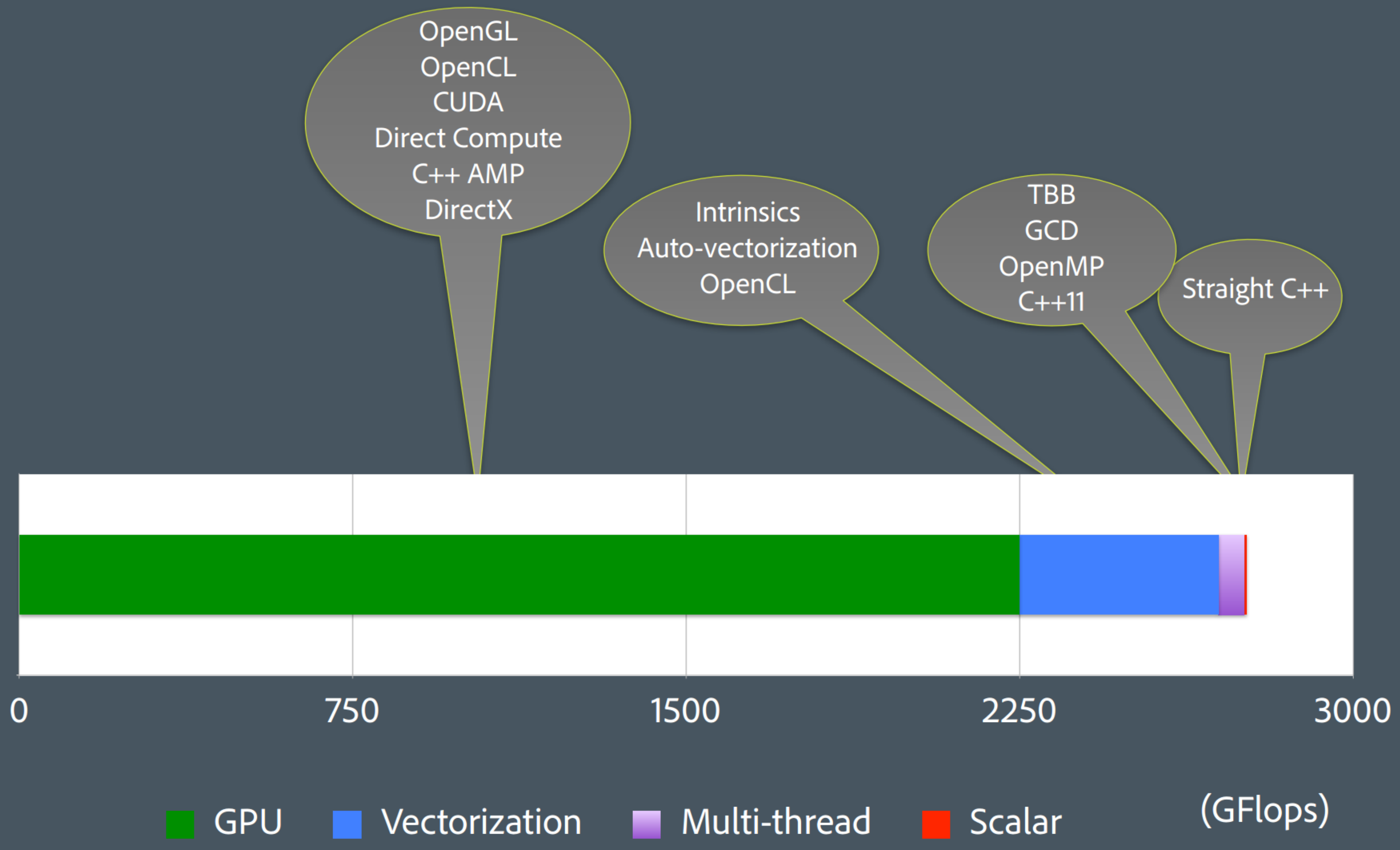
Development of cross-platform communication library in C++, with support for multiple scripting languages: architectural pitfalls.

By Oleg Iakushkin

PhD student from Saint Petersburg State University (SPBU)

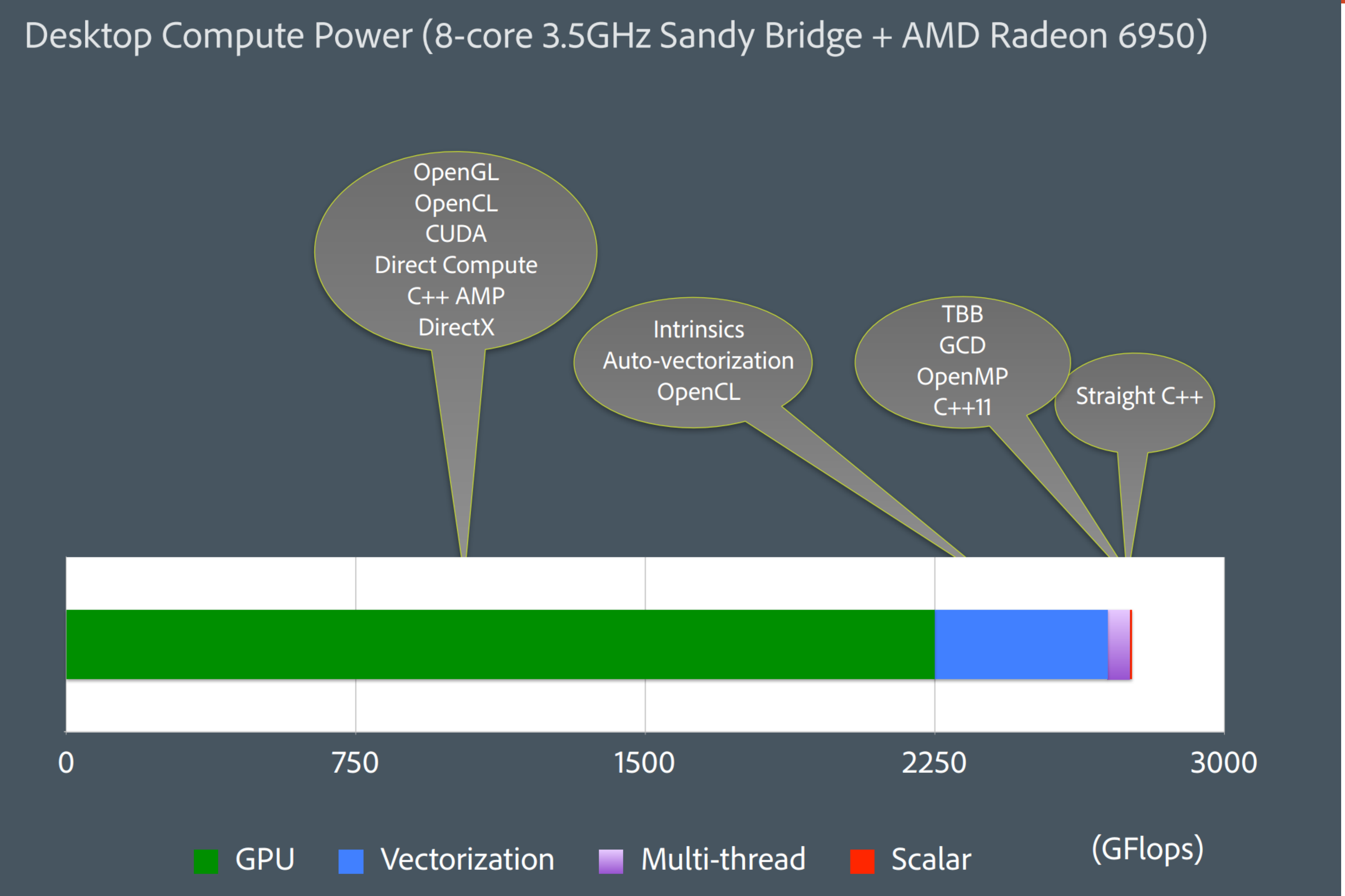
Why C++?

Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



Better Code: Concurrency
Sean Parent

Why C++?



Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

Src: Better Code: Concurrency
Sean Parent

NEC2015: Ian Bird discussion on CERN Compute requirements

Make your C++ code available – create portable API!

- C++ (SWIG, manual interop)

- C (manual interop)

Language selection C

When you create a library interface, you will most probably prefer to get as wide range of possible languages as possible. One might try to create a C interface as big teams do for libraries such ZeroMQ. Yet, I must recommend against it:

- It is extremely hard to change it during early development stages (in which it would change often).
- It will drag you away from C++ into C development – raw strings, void * pointers, no struct/class abstractions etc.
- C is great yet if your main language is C++ it will be hard to create sustainable C API on top of C++ codebase (or it will take lots of time)

Language selection C++

There are lots of code formatting styles for C++ such as Mozilla Developer guide on Coding style, Google C++ Style Guide, Joint Strike Fighter C++ Coding Standards and C++ Core Guidelines

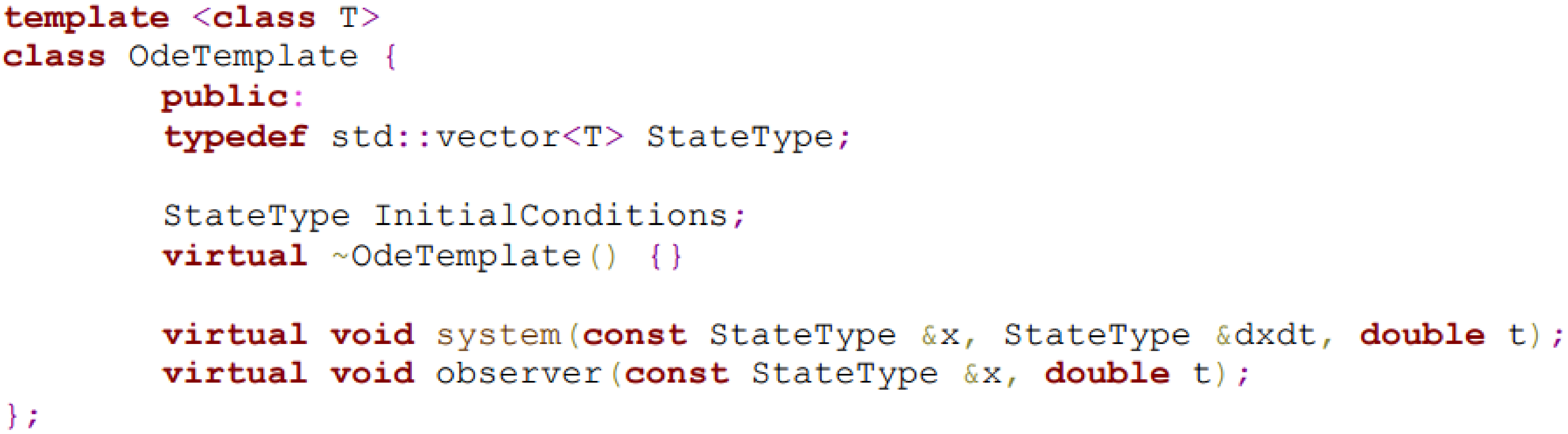
- https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style
- https://google-styleguide.googlecode.com/svn/trunk/cppguide.html
- http://www.stroustrup.com/JSF-AV-rules.pdf
- https://github.com/isocpp/CppCoreGuidelines

Passing Events

C++ has many ways to express an event model: C function pointer, functional object, lambda expression, std::function...

Wrappers that produce managed code tend to escape event code generation for most languages simply skipping related code.

Events via inheritance



```
template <class T>
class OdeTemplate {
        public:
        typedef std::vector<T> StateType;

        StateType InitialConditions;
        virtual ~OdeTemplate() {}

        virtual void system(const StateType &x, StateType &dxdt, double t);
        virtual void observer(const StateType &x, double t);
};
```

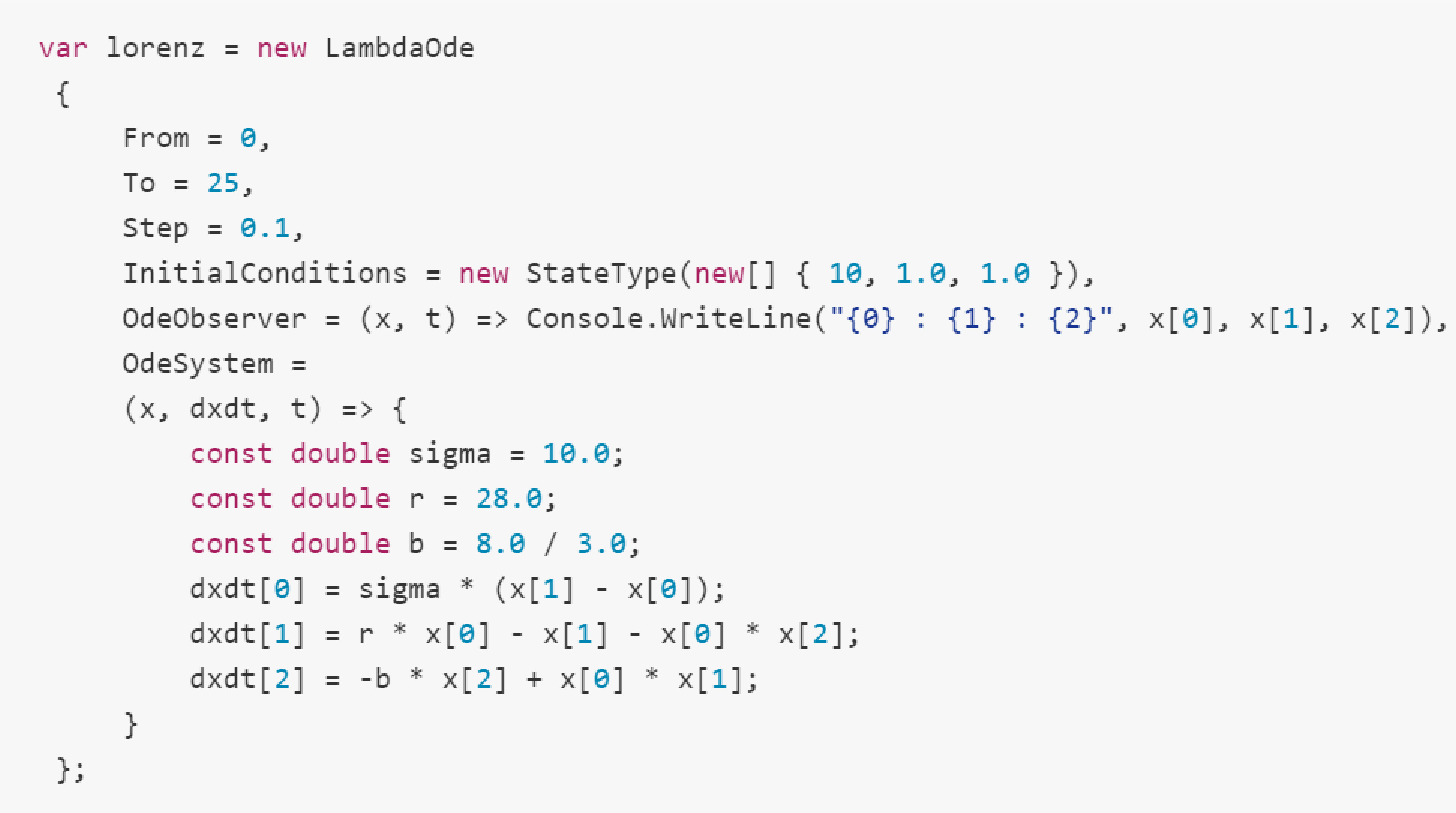
Templates give us pain

```
%include "std_complex.i"
%include "std_vector.i"

%{
#include "OdeProxy.h"
%}
%include "C++/OdeProxy.h"

%module(directors="1") Core

%feature("director") OdeProxy::OdeTemplate<double>;
%template(StateType) std::vector<double>;
%template(Ode) OdeProxy::OdeTemplate<double>;
%template(Solver) OdeProxy::SolverTemplate<double>;

%feature("director") OdeProxy::OdeTemplate<Complex>;
%template(ComplexStateType) std::vector<Complex >;
%template(ComplexOde) OdeProxy::OdeTemplate<Complex>;
%template(ComplexSolver) OdeProxy::SolverTemplate<Complex>;
```

How one can wrap it in target language

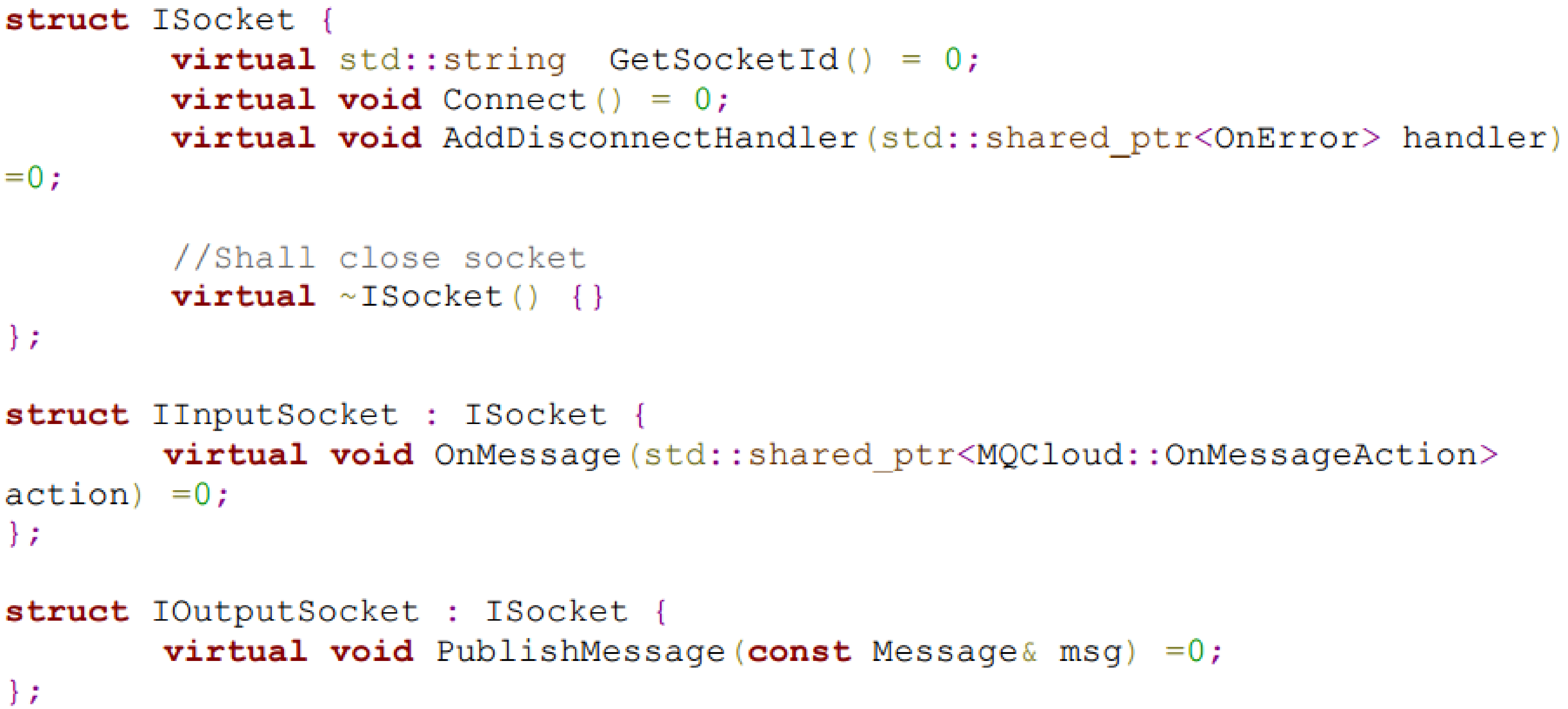
```
using System;

namespace OdeLibrary {
    public class ComplexLambdaOde : ComplexOde {
        public Action<ComplexStateType, ComplexStateType, double> OdeSystem {
get; set; }

        public Action<ComplexStateType, double> OdeObserver { get; set; }

        protected override void system(ComplexStateType x, ComplexStateType
dxdt, double t) {
            if (OdeSystem != null) {
                OdeSystem(x, dxdt, t);
            }
        }

        protected override void observer(ComplexStateType x, double t) {
            if (OdeObserver != null) {
                OdeObserver(x, t);
            }
        }
    }
}
```

C# Usage



```
var lorenz = new LambdaOde
 {
     From = 0,
     To = 25,
     Step = 0.1,
     InitialConditions = new StateType(new[] { 10, 1.0, 1.0 }),
     OdeObserver = (x, t) => Console.WriteLine("{0} : {1} : {2}", x[0], x[1], x[2]),
     OdeSystem =
     (x, dxdt, t) => {
         const double sigma = 10.0;
         const double r = 28.0;
         const double b = 8.0 / 3.0;
         dxdt[0] = sigma * (x[1] - x[0]);
         dxdt[1] = r * x[0] - x[1] - x[0] * x[2];
         dxdt[2] = -b * x[2] + x[0] * x[1];
     }
 };
```

Templates and Interfaces



```
struct ISocket {
        virtual std::string  GetSocketId() = 0;
        virtual void Connect() = 0;
        virtual void AddDisconnectHandler(std::shared_ptr<OnError> handler)
=0;

        //Shall close socket
        virtual ~ISocket() {}
};

struct IInputSocket : ISocket {
        virtual void OnMessage(std::shared_ptr<MQCloud::OnMessageAction>
action) =0;
};

struct IOutputSocket : ISocket {
        virtual void PublishMessage(const Message& msg) =0;
};
```

Avoid generic interfaces

```
template<typename TSocketBase>
struct IInputSocket : TSocketBase {

        virtual void OnMessage(std::shared_ptr<MQCloud::OnMessageAction>
action) =0;
};

template<typename TSocketBase>
struct IOutputSocket : TSocketBase {
        virtual void PublishMessage(const Message& msg) =0;
};
```

User space functions

The only fast, cross language way to pass user functions in SWIG is using inheritance. It is useful to provide utilities in end user language (like lambda/functional style events and async routines).

Conclusion

1. Pure C APIs with C++ backend make development iterations longer.
2. It is important to keep architecture as simple and bare bone C++ as possible.
3. Any standard library object can cause pain in one language or another (like for example std::complex).
4. Minimize header file includes required by your API.
5. Templates require special treatment and are not there in generated wrappers code.
6. Create special target language helper objects that could help integration of your library into users codebase.

C++ and SWIG can help adoption of your performant code

THANK YOU FOR YOUR ATTENTION