C++ 11/14
Writing modern high performance code

Jonathan Hollocombe
14th June 2016

Overview

- History & Motivation
- C++11/14
    - Smart Pointers
    - Type Inference
    - Functional Programming
    - R-value References
    - Other language constructs & libraries
- C++17

About C++

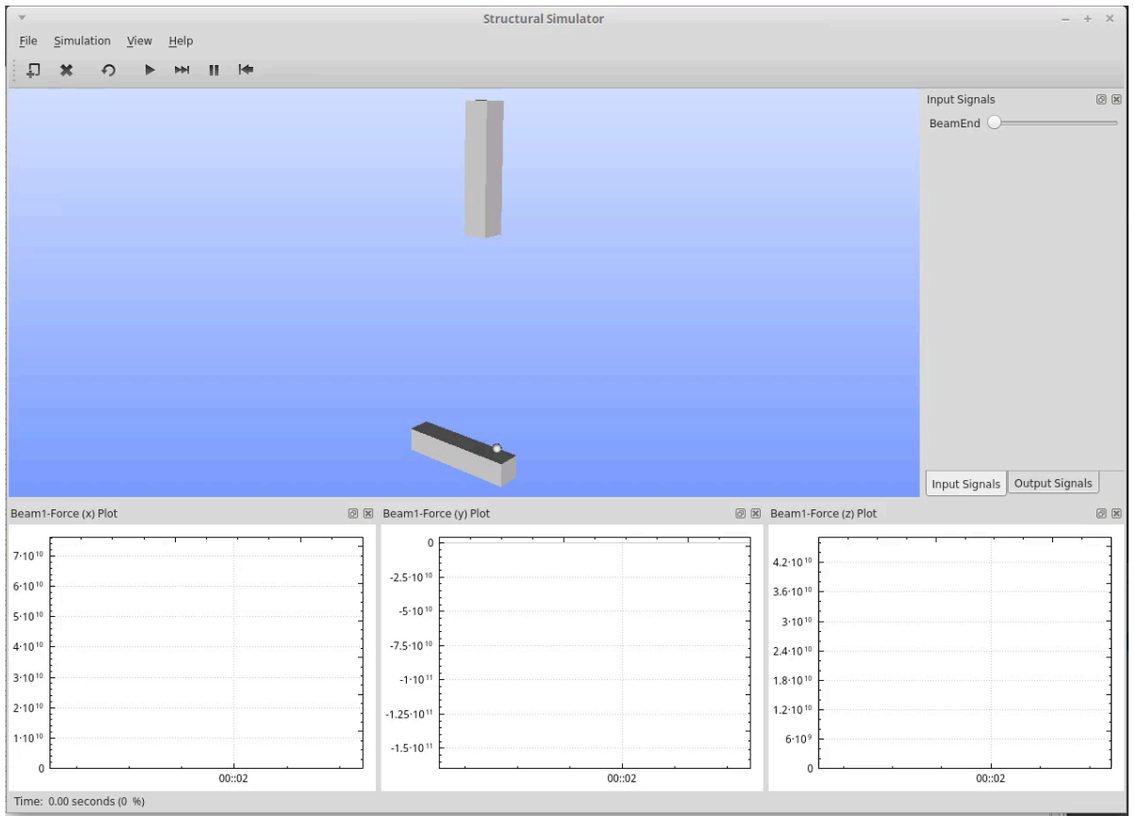
- First created in 1983 as a way of including Simula style abstractions into a systems level language
- Statically typed, multi-paradigm language
- Built on the idea of “zero cost” abstraction
- Encompasses the gamut from template meta-programming to machine code
- New C++ standards have introduced many new features to help modernise the language

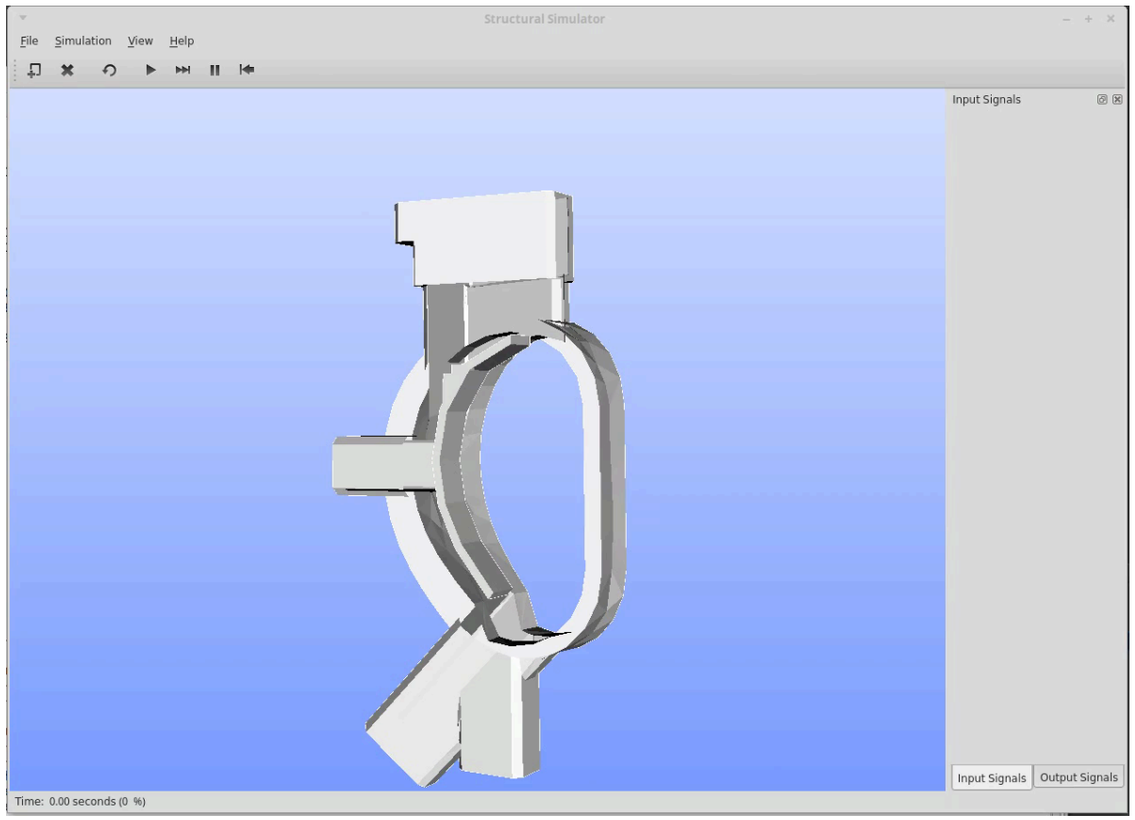
Why use C++?

- Efficient system level code with a more expressive & powerful language than C

- Low level extensions to high level languages (i.e. Python extension modules)

- Modern multi-paradigm language that can directly interface drivers, call GPU APIs, spawn system level threads, etc.





C++ project at CCFE

Structural Simulator project for DEMO remote maintenance (Qt5, VTK, SOFA, MkniX)

Modern C++

- C++11 available with gcc ≥4.8.1, clang ≥3.3
- C++14 available with gcc ≥5.0, clang ≥3.4
- Compile with option:
  `--std=c++11` or `--std=c++14`
  (`--std=c++0x` or `--std=c++1y` on older compilers)
- The examples in this presentation were compiled with clang 3.5
- All code examples are available on github (link at end)

Modern C++

- Smart Pointers
- Type Inference
- New Language Constructs
- New Standard Libraries
- Upcoming features (C++17)

Smart Pointers

- Memory management with-out garbage collection
- Reference counted pointers:
  `std::shared_pointer<Type>`
  `std::make_shared(…);`
- Unique ownership pointers:
  `std::unique_pointer<Type>`
  `std::make_unique(…); // C++14`
- No need for naked `new` or `delete`

```cpp
struct File {
    File(const std::string& file_name) : fid_(fopen(file_name.c_str(), "w")) {}
    ~File() { fclose(fid_); }
    bool is_open() { return fid_ != nullptr; }

private:
    FILE* fid_;
};

int main()
{
    File* file1 = new File{ "test1.txt" };

    if (!file1->is_open()) {
        throw std::logic_error("something went wrong");
    }

    std::unique_ptr<File> file2{ new File{ "test2.txt" } };              // C++11
    std::unique_ptr<File> file3 = std::make_unique<File>("test3.txt");  // C++14

    delete file1;

    return 0;
}
```

```cpp
struct SmartPointers {
    SmartPointers(const std::string& file_name)
    {
        unique_file_ = std::make_unique<File>(file_name);
        shared_file_ = std::make_shared<File>(file_name);
    }

    std::unique_ptr<File> transfer_ownership()
    {
        return std::move(unique_file_);
    }

    std::shared_ptr<File> share_ownership()
    {
        return shared_file_;
    }

private:
    std::unique_ptr<File> unique_file_;
    std::shared_ptr<File> shared_file_;
};

int main()
{
    SmartPointers pointers{ "test4.txt" };
    std::shared_ptr<File> shared = pointers.share_ownership();
    std::unique_ptr<File> unique = pointers.transfer_ownership();

    return 0;
}
```

Type Inference

- New use of keyword `auto`
- Type inference allows for flexibility while keeping compile-time type safety
- C++14 extends type inference to functions and lambdas

```cpp
auto auto_function() {
    return 1;
}

auto auto_function_with_return_type() -> double {
    return 1;
}

auto derived_types(int x, double y) -> decltype(x + y) {
    return x + y;
}

int main()
{
    auto vec = std::vector<double>{ 1, 2, 3, 4, 5 };    // variable declaration
    auto mult = [](double x, int p) { return x * p; };  // lambda type (C++11)
    auto square = [](const auto& x) { return x * x; };  // lambda type (C++14)

    for (auto& x : vec) {
        x = square(x);
    }

    for (const auto& x : vec) {
        std::cout << x << "\n";
    }

    std::cout << auto_function() << "\n";
    std::cout << auto_function_with_return_type() << "\n";
    std::cout << derived_types(1, 3.0) << "\n";

    return 0;
}
```

Functional Programming

- Introduction of lambdas and std::function allows for easier functional programming

- Combine lambdas with standard algorithms to generate expressive functional code

- Generic lambdas (C++14) allow for lambdas which can be used on multiple types

```cpp
struct Lambdas {
    Lambdas() {
        plain_func_ = [](int x) { return 2.0 * x; };
        bound_func_ = [this](int x) { return plain_func_(x); };
    }

private:
    std::function<double(int)> plain_func_;
    std::function<double(int)> bound_func_;
};

int main()
{
    int x = 0;

    auto ref_capture = [&x](int y) { x = y; };
    auto val_capture = [x](int y) { return x * y; };
    auto ref_all = [&](int y) { x = 2; };
    auto val_all = [=](int y) { return x * y; };

    auto type_inferred = [](auto n) { return n + 1; };

    static_assert(std::is_same<decltype(type_inferred(1.0)), double>::value, "");
    static_assert(std::is_same<decltype(type_inferred(1)), int>::value, "");

    return 0;
}
```

```cpp
struct Data {
    int index() const { return index_; }

private:
    int index_ = random_index();
};

int main()
{
    std::vector<std::shared_ptr<Data>> vec(10);

    std::for_each(vec.begin(), vec.end(), [](auto& el){
        el = std::make_shared<Data>();
    });

    std::sort(vec.begin(), vec.end(), [](auto& lhs, auto& rhs) {
        return lhs->index() < rhs->index();
    });

    auto is_even = [](const auto& el){ return el->index() % 2 == 0; };

    auto any_even = std::any_of(vec.begin(), vec.end(), is_even);

    decltype(vec) evens;
    if (any_even) {
        std::copy_if(vec.begin(), vec.end(), std::back_inserter(evens), is_even);
    }

    std::for_each(evens.begin(), evens.end(), [](const auto& el) {
        std::cout << el->index() << "\n";
    });

    return 0;
}
```

```cpp
struct MyData {
    double* begin() { return &data_[0]; }
    double* end() { return &data_[Length]; }

private:
    constexpr static size_t Length = 3;
    double data_[Length] = { 0.1, 0.2, 0.3 };
};

namespace std {
    template <> double* begin(MyData& data) { return data.begin(); }
    template <> double* end(MyData& data) { return data.end(); }
}

int main()
{
    auto sum = [](auto& v) { return std::accumulate(std::begin(v), std::end(v),
0.0, std::plus<double>()); };

    std::vector<double> vec = { 0.1, 0.2, 0.3 };
    double array[] = { 0.1, 0.2, 0.3 };
    MyData data;

    std::cout << sum(vec) << std::endl;
    std::cout << sum(array) << std::endl;
    std::cout << sum(data) << std::endl;

    return 0;
}
```

R-value References

- New reference type added (&&)
- Allows for the move semantics needed for `std::unique_ptr`
- Move constructors and move operators remove copy overhead on large objects
- STL containers use move instead of copy (the problem with the now deprecated `std::auto_ptr`)

Reference Types

```cpp
std::string f(int& x) { return "l-value"; }
std::string f(int&& x) { return "r-value"; }

int g() { return 1; }

void h(int& x) {
    std::cout << f(x) << std::endl;                         // l-value
}

void i(int&& x) {
    std::cout << f(x) << std::endl;                         // l-value
    std::cout << f(std::forward<int>(x)) << std::endl;      // r-value
}

int main()
{
    int x = 1;

    std::cout << f(x) << std::endl;                         // l-value
    std::cout << f(g()) << std::endl;                       // r-value
    std::cout << f(std::move(x)) << std::endl;              // r-value

    h(x);
    i(std::move(x));

    return 0;
}
```

Avoiding Copies

```
struct BigData {
    BigData() : vec(1000000) {}
    BigData(const BigData& other) {
        std::cout << "i've been copied\n";
        vec = other.vec;
    }
    BigData(BigData&& other) {
        std::cout << "i've been moved\n";
        vec = std::move(other.vec);
    }

private:
    std::vector<int> vec;
};

BigData&& g(BigData&& vec)
{
    return std::move(vec);
}

BigData f()
{
    BigData data;
    return g(std::move(data));
}

int main()
{
    auto data = f();
    return 0;
}
```

Moveable Resources

```
struct File {
    File(const std::string& file_name) : fid_(fopen(file_name.c_str(), "w")) {}
    ~File() { fclose(fid_); }

    // Move constructor
    File(File&& other) {
        fid_ = other.fid_;
        other.fid_ = nullptr;
    }

    // Make this non-copyable
    File(const File&) = delete;
    File& operator=(const File&) = delete;

    FILE* fid() { return fid_; }

private:
    FILE* fid_ = nullptr;
};

int main()
{
    File file{ "myfile.txt" };
    File new_file = std::move(file);

    std::cout << file.fid() << std::endl;
    std::cout << new_file.fid() << std::endl;

    std::vector<File> files;
    files.emplace_back("myfile.txt");
    files.push_back(std::move(File{"myfile.txt"}));

    return 0;
}
```

New Language Constructs

- ranged based for loops
- enum class
- override
- constexpr
- uniform initialisation and initialiser lists
- nullptr
- user defined literals
- static_assert
- variadic templates

Range based for loops

```cpp
int main()
{
    std::vector<int> vec = { 0, 1, 2, 3, 4, 5 };

    // C++03
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << "\n";
    }

    // C++11
    for (const auto& x : vec) {
        std::cout << x << "\n";
    }

    return 0;
}
```

Enum Class

```cpp
enum class Color : unsigned short {
    Red, Yellow, Green
};

std::string to_string(const Color& color)
{
    switch (color) {
        case Color::Red: return "Red";
        case Color::Yellow: return "Yellow";
        case Color::Green: return "Green";
    }
}

int main()
{
    Color color = Color::Red;

    std::cout << to_string(color) << std::endl;

    return 0;
}
```

Override Keyword

```
struct Base {
    virtual ~Base() {}
    virtual void method() = 0;
    virtual int const_method(int x) const = 0;
};

struct Derived : public Base {
    void method() override {}
    int const_method(int x) const override { return x; }
};

int main()
{
    auto x = std::make_unique<Derived>();
    x->method();
    x->const_method(1);

    return 0;
}
```

Constexpr

```
constexpr double gen_pi()
{
    const int N = 400;
    int count = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            auto x = static_cast<double>(i) / N;
            auto y = static_cast<double>(j) / N;
            auto len = x*x + y*y;
            if (len < 1) ++count;
        }
    }
    return (count * 4.0) / (N * N);
}

constexpr auto PI = gen_pi();

int main()
{
    static_assert(static_cast<int>(PI * 10) / 10.0 == 3.1);

    std::cout << PI << std::endl;

    return 0;
}
```

uniform initialisation & initialiser lists

```
struct MyList {
    MyList(std::vector<int> vec) : vec_{ vec } {}
    MyList(std::initializer_list<int> lst) : vec_{ lst } {}

private:
    std::vector<int> vec_;
};

int main()
{
    std::vector<int> vec = { 1, 2, 3, 4, 5 };

    MyList list_from_vector{ vec };
    MyList list_from_initlist{ 1, 2, 3, 4 };

    return 0;
}
```

user defined literals

```
struct Distance {
    constexpr Distance(double metres) : metres_(metres) { }
    constexpr Distance operator/ (const Distance& rhs) { return Distance{ metres_ /
rhs.metres_ }; }
    std::string to_string() const { return std::to_string(metres_) + "m"; }
private:
    double metres_;
};

std::ostream& operator<<(std::ostream& out, const Distance& obj)
{
    out << obj.to_string();
    return out;
}

constexpr Distance operator"" _m(unsigned long long metres)
{
    return Distance{ static_cast<double>(metres) };
}

constexpr Distance operator"" _km(unsigned long long metres)
{
    return Distance{ 1E3 * static_cast<double>(metres) };
}

int main()
{
    auto x = 100_m;
    auto y = 10_km;

    std::cout << x / y << std::endl;

    return 0;
}
```

static_assert

```cpp
template <typename T>
struct Coord {
    static_assert(std::is_arithmetic<T>::value, "template argument T must be
arithmetic");
    Coord(T x, T y) : x_(x), y_(y) {}
    std::string to_string() const { return "[" + std::to_string(x_) + ", " +
std::to_string(y_) + "]"; }

private:
    T x_, y_;
};

template <typename T>
std::ostream& operator<<(std::ostream& out, const Coord<T>& coord)
{
    out << coord.to_string();
    return out;
}

int main()
{
    Coord<double> c{ 0.1, 2.0 };
    // Coord<std::string> d{ "a", "b" };

    std::cout << c << std::endl;

    return 0;
}
```

variadic templates

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

New Standard Libraries

- <tuple>
- <regex>
- <random>
- <functional>
- <chrono>

C++17 Highlights

- nested namespace definitions:
  `namespace A::B::C {}`

- <filesystem>

- parallel algorithms
  `std::sort(std::par, v.begin(), v.end())`

- `std::any` and `std::optional`

- more to be added

Thank you for listening

Links

- Code: https://github.com/jholloc/cxxtalk
- C++ Reference: http://en.cppreference.com
- Boost: http://www.boost.org/

Any questions?