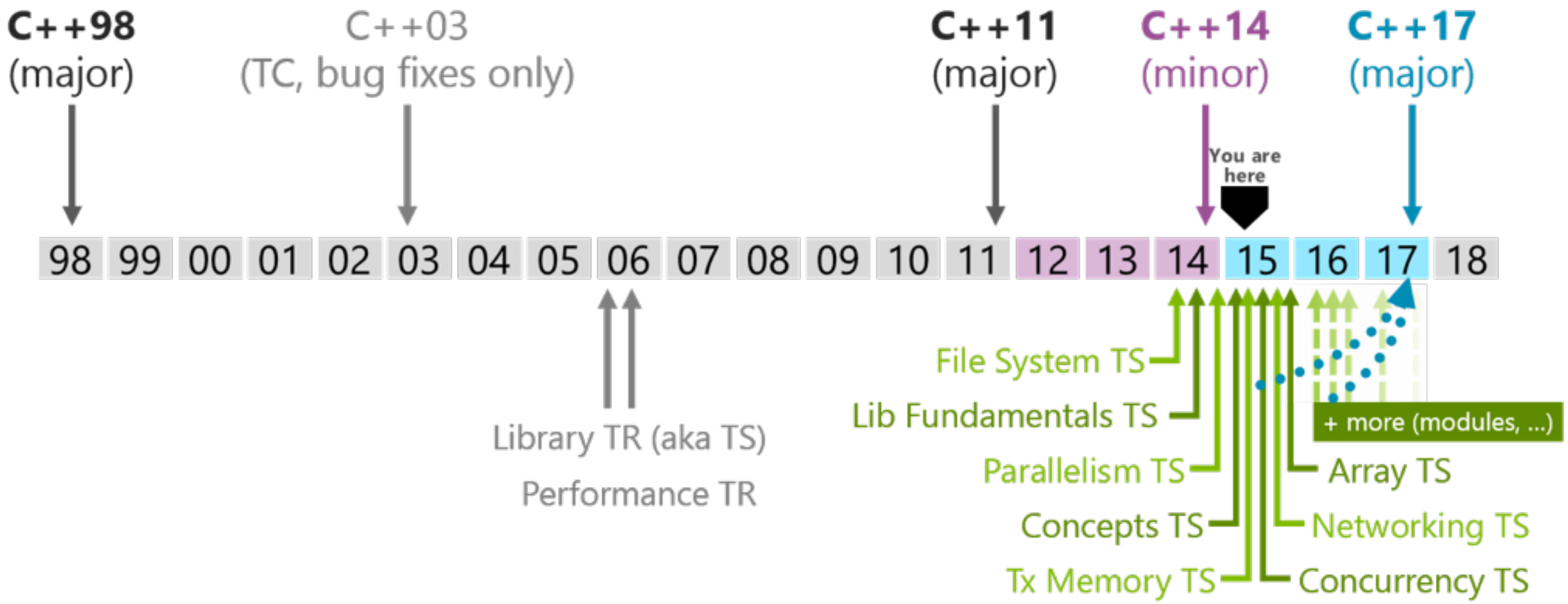
Introduction to Modern C++

Alejandro Isaza

al@isaza.ca

@aleph7



https://isocpp.org/std/status

Reference

- http://en.cppreference.com
- http://www.cplusplus.com/reference/

Part 1

Introduction

Types

```
// An integer variable
int myVariable = 3;

// A boolean variable
bool myBoolean = true;

// A single-precision floating-point variable
float myFloat = 1.618f;

// A double-precision floating-point variable
double myDouble = 3.142;

// A fixed-size integer variable
int32_t my32BitVariable = 0b010101010;

// An unsigned integer variable (stay away)
unsigned int myUnsignedVariable = 0xC001u;
```

auto

```
// An integer variable
auto myVariable = 3;

// A boolean variable
auto myBoolean = true;

// A single-precision floating-point variable
auto myFloat = 1.618f;

// A double-precision floating-point variable
auto myDouble = 3.142;

// A fixed-size integer variable
auto my32BitVariable = int32_t{0b010101010};

// An unsigned integer variable (stay away)
auto myUnsignedVariable = 0xC001u;
```

Constants

```
// A constant
const auto myConstant = 3;

// A constant expression
constexpr auto myConstantExpression = 7;
```

Constants

```
const auto size = 3;
std::array<int, size> array2; // Maybe error
```

```
constexpr auto size = 3;
std::array<int, size> array; // Ok
```

References

```
int main(int argc, const char* argv[]) {
    auto a = 1;
    auto& refToA = a;
    refToA = 2;

    std::cout << a << std::endl;
    return 0;
}
```

References

```
int main(int argc, const char* argv[]) {
    auto a = 1;
    const auto& refToA = a;
    refToA = 2; // Error: Read-only variable is not assignable

    std::cout << a << std::endl;
    return 0;
}
```

References

```
int main(int argc, const char* argv[]) {
    auto a = 1;
    auto b = 4;
    auto& refToA = a;
    refToA = b;

    std::cout << a << std::endl;
    return 0;
}
```

References

```
void increase(int& v) {
    v += 1;
}

int main(int argc, const char* argv[]) {
    auto a = 1;
    increase(a);

    std::cout << a << std::endl;
    return 0;
}
```

References

```
int& getValue() {
    int a = 1;
    return a; // Warning: Reference to stack memory returned
}
```

Pointers

```
int main(i[illegible] const char* argv[]) {
    auto[illegible] = 1;
    aut[illegible]* refToA[illegible];
    *r[illegible]fToA = [illegible];

    std[illegible]:c[illegible]t << a [illegible]< std::endl;
    retu[illegible]0;
}
```

Pointers

```
const char* message = "Hello";

int main(int argc, const char* argv[]) {
    std::cout << message << std::endl;
    return 0;
}
```

Pointers

```cpp
void print(int* p) {
    if (p)
        std::cout << *p << std::endl;
}

int main(int argc, const char* argv[]) {
    int* pointer = nullptr;
    print(pointer);

    int a = 3;
    pointer = &a;
    print(pointer);

    return 0;
}
```

“Anybody who comes to you and says he has a perfect language is either naïve or a salesman”

–Bjarne Stroustrup

Structures

```
// Declare a new type
struct Company {
    std::string name;
    int numberOfEmployees;
};
```

Structures

```
// Declare a new type
struct Company {
    std::string name;
    int numberOfEmployees;
};

// Instantiate the type in a new variable
auto company = Company{"Pied Piper", 4};
```

Structures

```
// Declare a new type
struct Company {
    std::string name;
    int numberOfEmployees;
};

// Instantiate the type in a new variable
auto company = Company{"Pied Piper", 4};

void addEmployee() {
    // Access an element of the structure
    company.numberOfEmployees += 1;
}
```

Functions

```
int fib(int n) {
    if (n == 0 || n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Functions

```
inline constexpr auto fib(int n) noexcept {
    if (n == 0 || n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

std::array<int, fib(4)> a;
```

Classes

```
class Person {
public:
    const std::string& name() const {
        return _name;
    }
    void setName(std::string name) {
        _name = name;
    }

private:
    std::string _name;
};
```

Classes

```
class Person {
public:
    Person() = default;
    Person(std::string name) : _name(name) {}

    const std::string& name() const {
        return _name;
    }
    void setName(std::string name) {
        _name = name;
    }

private:
    std::string _name;
};
```

Includes

```
#ifndef PERSON_H
#define PERSON_h

#include "Address.h"
#include <string>

class Person {
public:
    //...

private:
    std::string _name;
};

#endif
```

Includes

```cpp
#pramga once

#include "Address.h"
#include <string>

class Person {
public:
    //...

private:
    std::string _name;
};
```

Initializer lists

```
// Initialize a vector with a list
std::vector<int> v{1, 3, 5, 7};

// Warning: this is like {1, 1, 1}
std::vector<int> v(3, 1);
```

Initializer lists

```
#include <initializer_list>
#include <vector>

class Collection {
public:
    std::vector<double> vector;
    Collection(std::initializer_list<double> list) : vector(list) {}

    //..
};
```

Forward Declarations

```
class Address;

class Person {
public:
    const std::string& name() const {
        return _name;
    }
    void setName(std::string name) {
        _name = name;
    }

private:
    std::string _name;
    std::unique_ptr<Address> _address;
};
```

Namespaces

```cpp
namespace mcw {

class Address;

class Person {
public:
    const std::string& name() const {
        return _name;
    }
    void setName(std::string name) {
        _name = name;
    }

private:
    std::string _name;
    std::unique_ptr<Address> _address;
};

} // namespace
```

Templates

```
inline constexpr auto add(int a, int b) {
    return a + b;
}

inline constexpr auto add(double a, double b) {
    return a + b;
}


template<typename T>
inline constexpr auto add(T a, T b) {
    return a + b;
}
```

Templates

```
template <typename T>
class Vector2D {
public:
    Vector2D() = default;
    Vector2D(T x, T y) : _x(x), _y(y) {}

    T x() const { return _x; }
    T y() const { return _y; }

    void setX(T x) { _x = x; }
    void setY(T y) { _y = y; }

private:
    T _x = 0;
    T _y = 0;
};

auto v = Vector2D<double>{1, 1};
```

NAVY
100
CF-01
F-35

Move semantics

```
int process(std::vector<int> v) {
    //...
}

int main(int argc, const char* argv[]) {
    std::vector<int> v{...};
    std::cout << process(std::move(v)) << std::endl;
    return 0;
}
```

Other things

- #defines are a C thing, don’t use them
- NULL is a C thing, don’t use it
- zero-terminated strings … don’t use them
- Consider writing only headers (use inline)

C++ is not C

Hello World!

```cpp
#include <iostream>
#include <string>

int main(int argc, const char* argv[]) {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The Standard Library

std::string

```
#include <string>

auto getMessage() {
    auto message = std::string("hello");
    message.append(" ");
    message += "world";
    return message;
}

auto getName() {
    using namespace std::string_literals;
    auto name = "Alejandro"s;
    return name;
}
```

std::vector

```
#include <cassert>
#include <vector>

auto printElements() {
    auto v = std::vector<int>{1, 1, 2, 3, 5};
    assert(v.size() == 5);

    v.push_back(8);
    assert(v[5] == 8);

    for (auto& element : v)
        std::cout << element << "\n";
}
```

std::map

```cpp
#include <map>

auto colors = std::map<std::string, uint32_t>{
    {"red", 0xff0000},
    {"green", 0x00ff00},
    {"blue", 0x0000ff},
};

auto addColor(const std::string& name, uint32_t value) {
    colors[name] = value;
}

auto getColor(const std::string& name) {
    return colors[name];
}
```

Other Containers

- std::array
- std::deque
- std::list
- std::queue
- std::priority_queue
- std::stack
- std::set, std::multiset, std::unordered_set, std::unordered_multimap
- std::map, std::multimap, std::unordered_map, std::unordered_multimap

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

auto generateEven(int n) {
    auto values = std::vector<int>{};
    auto x = 0;
    auto gen = [&x]() { return x += 2; };
    std::generate_n(std::inserter(values, std::begin(values)), n, gen);
    return values;
}

auto isEven(int v) { return v % 2 == 0; }

auto checkEven(const std::vector<int>& vector) {
    return std::all_of(std::begin(vector), std::end(vector), isEven);
}

int main(int argc, const char * argv[]) {
    auto values = generateEven(5);
    std::copy(std::begin(values), std::end(values),
              std::ostream_iterator<int>(std::cout, " "));
    std::cout << "\nAll values are "
              << (checkEven(values) ? "even" : "not even")
              << std::endl;
    return 0;
}
```

Streams

```cpp
#include <iostream>
#include <fstream>

void write(const char* fileName) {
    std::ofstream ofs(fileName);
    ofs << "Hello, world!" << std::endl;
}

std::string read(const char* fileName) {
    std::ifstream ifs(fileName);
    std::string line;
    std::getline(ifs, line);
    return line;
}

int main(int argc, const char * argv[]) {
    static const auto fileName = "test.txt";
    write(fileName);
    std::cout << "file contents: " << read(fileName) << std::endl;
    return 0;
}
```

Iterators

```
int main(int argc, const char* argv[]) {
    std::vector<int> v{1, 3, 5, 7};
    for (auto it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << std::endl;
    }
    return 0;
}
```

Iterators

```
int main(int argc, const char* argv[]) {
    std::vector<int> vector{1, 3, 5, 7};
    for (auto& value : vector) {
        std::cout << value << std::endl;
    }
    return 0;
}
```

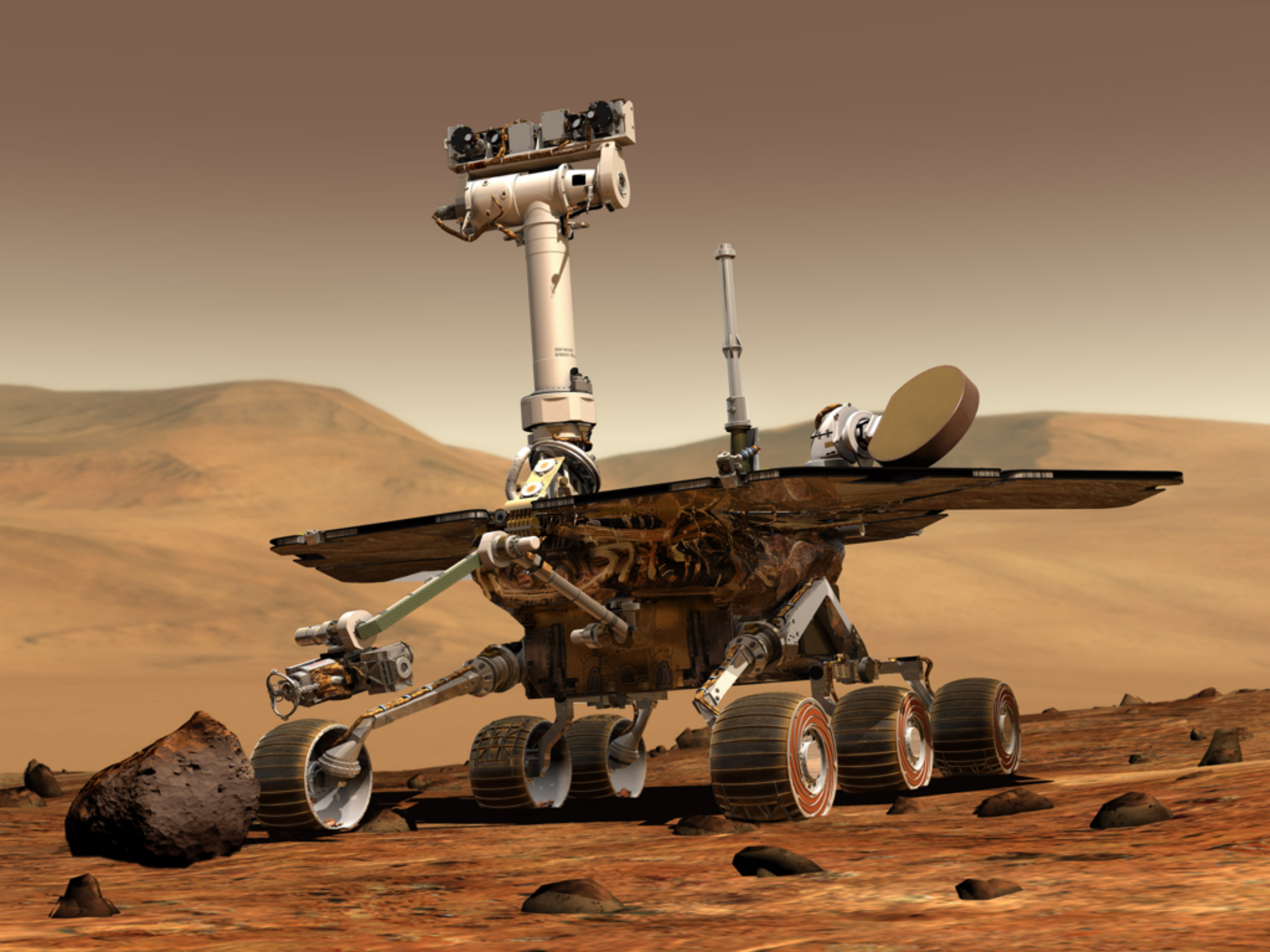
Iterators

```
int main(int argc, const char* argv[]) {
    std::vector<int> v{1, 3, 5, 6, 7};
    for (auto it = v.begin(); it != v.end(); ++it) {
        auto next = std::next(it);
        if (*next % 2 == 0)
            std::cout << *it << std::endl;
    }
    return 0;
}
```

Iterators

```
template <typename It>
void printIfNextIsEven(It begin, It end) {
    while (begin != end) {
        auto next = std::next(begin);
        if (*next % 2 == 0)
            std::cout << *begin << std::endl;
        ++begin;
    }
}

int main(int argc, const char* argv[]) {
    std::list<int> l{1, 2, 3, 5, 6};
    printIfNextIsEven(l.begin(), l.end());
    return 0;
}
```



Algorithms

http://en.cppreference.com/w/cpp/header/algorithm

Other stuff

```cpp
#include <cmath>
#include <iostream>

int main(int argc, const char* argv[]) {
    auto a = -1;
    std::cout << std::abs(a) << std::endl;

    auto b = -1.2f;
    std::cout << std::abs(b) << std::endl;

    auto c = -M_PI;
    std::cout << std::abs(c) << std::endl;

    return 0;
}
```

boost

http://www.boost.org/doc/libs/

UI/Graphics

http://libcinder.org

http://www.sfml-dev.org

http://www.qt.io/developers/

More Stuff

https://fffaraz.github.io/awesome-cpp/

What’s next

- Part 1: Introduction
- Part 2: Compilation
- Part 3: Writing a class
- Part 4: Unit tests
- Part 5: exceptions
- Part 6: Smart pointers
- Part 7: lambdas
- Part 8: templates
- Part 9: concurrency
- Part 10: Coding time

Part 2

Compiling

Compilers

- C++11 is mandatory
  - g++ 4.7.2
  - clang++ 3.0 (Xcode 5)
  - VS 2014
- C++14 is recommended
  - g++ 4.9
  - clang++ 3.4 (Xcode 6)
  - VS 2015

Compilers status

- http://gcc.gnu.org/projects/cxx1y.html
- http://clang.llvm.org/cxx_status.html
- http://blogs.msdn.com/b/vcblog/archive/2015/04/29/c-11-14-17-features-in-vs-2015-rc.aspx

Code Time

Understanding Error Messages

Warning Flags

- -Wall
- -Wpedantic
- -Wextra
- -Werror

https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Part 3

Writing a class

The Vector Class

```cpp
#pragma once

namespace mcw {

class Vector2D {
public:
    using Type = double;

public:
    // Constructors

    // Accessors

    // Operators

private:
    Type _x = 0;
    Type _y = 0;
};

} // namespace mcw
```

Operator Overloading

```
template <typename T>
class Vector2D {
public:
    //...

    Vector2D& operator+=(const Vector2D& rhs) {
        _x += rhs._x;
        _y += ths._y;
        return *this;
    }
private:
    T _x = 0;
    T _y = 0;
};

auto v1 = Vector2D{1, 1};
auto v2 = Vector2D{-1, -1};
v2 += v1;
```

Operator Overloading

```
template <typename T>
class Vector2D {
public:
    //...

private:
    T _x = 0;
    T _y = 0;
};


inline Vector2D operator+(const Vector2D& lhs, const Vector2D& rhs) {
    return Vector2D{lhs.x() + rhs.x(), lhs.y() + rhs.y()};
}
```

A pure abstract class

Shape.h

```
#pragma once
#include "Vector2D.h"

namespace mcw {

class Shape {
public:
    virtual ~Shape() = default;
    virtual bool hitTest(const Vector2D& point) const = 0;
};

} // namespace mcw
```

The Circle Class

Circle.h

```
#pragma once
#include "Shape.h"

namespace mcw {

class Circle : public Shape {
public:
    Circle(Vector2D center, double r);

    bool hitTest(const Vector2D& point) const override;

private:
    Vector2D _center;
    double _radius;
};

} // namespace mcw
```

The Circle Class

Circle.cpp

```
#include "Circle.h"

namespace mcw {

Circle::Circle(Vector2D center, double r)
: _center(center), _radius(r) {}

bool Circle::hitTest(const Vector2D& point) const {
    const auto delta = point - _center;
    return delta.magnitudeSquared() <= _radius * _radius;
}

} // namespace mcw
```

Code Time

Part 4

Unit Tests

Boost Unit Test Framework

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE "tempo"
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(truth) {
    BOOST_CHECK(true);
}
```

http://www.boost.org/doc/libs/1_58_0/libs/test/doc/html/utf.html

Testing tools

- BOOST_WARN
- BOOST_CHECK
- BOOST_REQUIRE
- BOOST_CHECK_EQUAL
- BOOST_CHECK_CLOSE
- BOOST_CHECK_SMALL
- BOOST_CHECK_EXCEPTION
- BOOST_CHECK_NO_THROW
- BOOST_CHECK_NE, GT, GE, LT, LE

Code Time

Part 5

Exceptions

Throwing

```
Vector2D normalize(const Vector2D& v) {
    auto norm = v.norm();
    if (norm == 0)
        throw std::invalid_argument("can't normalize zero vector");
    return {v.x / norm, v.y / norm};
}
```

Exception Types

- std::exception
  - std::logic_error
    - std::invalid_argument
    - std::length_error
  - std::runtime_error
    - std::overflow_error
    - std::system_error

Custom Exceptions

```
#include <stdexcept>

class MyError : public std::logic_error {
public:
    explicit MyError(const std::string& what_arg) : std::logic_error(what_arg) {}
    explicit MyError(const char* what_arg) : std::logic_error(what_arg) {}
};
```

Catching

```
int main(int argc, const char* argv[]) {
    int* numbers = nullptr;
    try {
        numbers = new int[1'000'000'000'000'000];
        std::cout << "Success" << std::endl;
    } catch (std::bad_alloc& e) {
        std::cerr << "Failure: " << e.what() << std::endl;
    }
}
```

Exception Safety

```
void generate(double frequency, int size) {
    static const double fs = 44100;
    int* numbers = new int[size];

    if (frequency == 0)
        throw std::invalid_argument("frequency can't be zero");

    for (int i = 0; i < size; i += 1) {
        const auto t = i / fs;
        numbers[i] = std::sin(2*M_PI*frequency*t);
    }

    // ...
    delete numbers;
}
```

Exception Safety

```
void generate(double frequency, int size) {
    static const double fs = 44100;
    int* numbers = new int[size];

    if (frequency == 0)
        throw std::invalid_argument("frequency can't be zero"); //< Leak

    for (int i = 0; i < size; i += 1) {
        const auto t = i / fs;
        numbers[i] = std::sin(2*M_PI*frequency*t);
    }

    // ...
    delete numbers;
}
```

noexcept

```cpp
inline int add(int a, int b) noexcept {
    return a + b;
}

class Vector2D {
public:
    Vector2D(double x, double y) noexcept : _x(x), _y(y) {}

private:
    double _x;
    double _y;
};
```

Part 6

Smart Pointers

Avoid new/delete

```
int main  nt argc   onst char* argv[]) {
    aut  p = ne   e  on{};

    //  ..

    dele    p;
}
```

std::unique_ptr

```
int main(int argc, const char* argv[]) {
    auto p = std::make_unique<Person>();
    p->setName("Al");
    // ...
}
```

std::unique_ptr

```
auto p = std::make_unique<Person>();
auto q = std::move(p);
```

std::shared_ptr

```cpp
struct Company {
    std::string name;
    int numberOfEmployees;
    std::shared_ptr<Person> ceo;
};

int main(int argc, const char* argv[]) {
    auto company = Company{"Pied Piper", 4};
    company.ceo = std::make_shared<Person>("Richard");
    return 0;
}
```

std::weak_ptr

```
auto person = std::make_shared<Person>();
auto weakPerson = std::weak_ptr<Person>(person);

// ...
auto strongPerson = weakPerson.lock();
if (strongPerson) {
    // ...
}
```

Part 7

Lambdas

A Lambda

```
auto lambda = [](){};
```

A Lambda

```
auto lambda = [](){ return 42; };
```

A Lambda

```cpp
auto frequency = 440;
auto lambda = [frequency](){ return std::sin(2*M_PI*frequency); };
```

A Lambda

```
auto frequency = 440.0;
auto lambda = [frequency](double phase) {
    return std::sin(2*M_PI*frequency + phase);
};

// Call it
lambda(0.0);
```

Capture Lists

```
// Capture a by value, b by reference
auto lambda1 = [a, &b](){};

// Automatic capture by value
auto lambda2 = [=](){};

// Automatic capture by reference
auto lambda3 = [&](){};
```

Function Objects

```
#include <functional>

void generate(std::function<double ()> f, int n) {
    for (int i = 0; i < n; i += 1)
        std::cout << f() << "\n";
}
```

Function Objects

```
#include <functional>

void generate(std::function<double ()> f, int n) {
    for (int i = 0; i < n; i += 1)
        std::cout << f() << "\n";
}

int main(int argc, const char* argv[]) {
    int i = 0;
    generate([&i]() { return i++; }, 20);
}
```

Part 8

Templates

A Generic Function

```cpp
inline constexpr auto add(int a, int b) {
    return a + b;
}

inline constexpr auto add(double a, double b) {
    return a + b;
}


template<typename T>
inline constexpr auto add(T a, T b) {
    return a + b;
}
```

A Generic Class

```cpp
template <typename T>
class Vector2D {
public:
    Vector2D() = default;
    Vector2D(T x, T y) : _x(x), _y(y) {}

    T x() const { return _x; }
    T y() const { return _y; }

    void setX(T x) { _x = x; }
    void setY(T y) { _y = y; }

private:
    T _x = 0;
    T _y = 0;
};

auto v = Vector2D<double>{1, 1};
```

Forwarding References

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

static_assert

```cpp
#include <type_traits>

template<typename T>
inline constexpr auto add(T a, T b) {
    static_assert(std::is_arithmetic<T>(), "T has to be an arithmetic type");
    return a + b;
}
```

Template Metaprogramming

```
template<typename LHS , typename RHS>
struct less : public std::integral_constant<bool, sizeof(LHS) < sizeof(RHS)> {};

template<typename LHS , typename RHS>
struct min {
    using type = typename std::conditional<less<LHS,RHS>::value, LHS, RHS>::type;
};

min<int16_t, int32_t>::type number = 0;
```

Part 9

Concurrency

std::atomic

```
std::atomic<int> counter;

int f() {
    counter++;
    return counter.load();
}
```

Mutexes

```
#include <mutex>

std::mutex myMutex;

void save() {
    myMutex.lock();
    // use shared resource
    myMutex.unlock();
}
```

Mutexes

```
#include <mutex>

std::mutex myMutex;

void save() {
    std::lock_guard<std::mutex> lock(myMutex);
    // Use shared resource
}
```

std::async

```cpp
#include <future>

int start() {
    auto future1 = std::async(std::launch::async, []{
        // Do chunk of work
        return 0;
    });
    auto future2 = std::async(std::launch::async, []{
        // Do another chunk of work
        return 0;
    });
    return future1.get() + future2.get();
}
```

Threads

```cpp
#include <thread>

void run() {
    std::thread thread([]() {
        // Do work
    });
    thread.join();
}
```

A Piece of Code

```cpp
std::shared_ptr<Widget> getWidget(int id) {
    static std::map<int, std::weak_ptr<Widget>> cache;
    static std::mutex m;

    std::lock_guard<std::mutex> hold(m);
    auto sp = cache[id].lock();
    if (!sp) cache[id] = sp = loadWidget(id);
    return sp;
}
```

https://www.youtube.com/watch?v=jK1_3RbEwLE

Part 10

Code Time