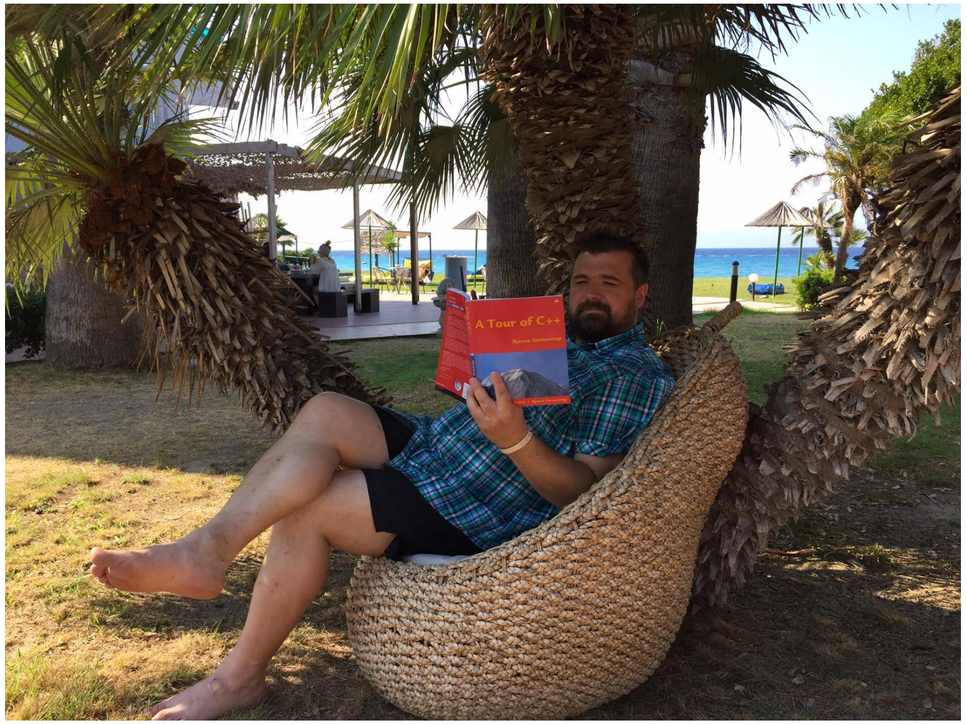
A Tour of Modern C++

with Olve Maudal



Instructor notes for live coding

2.5 hour lecture for C++ students at Høyskolen Kristiania, Oslo

Tuesday, Feb 27, 2018

Disclaimer

These are my personal notes that I use as a script while live coding and explaining modern C++. The notes by themselves probably do not make much sense for others, but I don't mind sharing them anyway.

Acknowledgement

Many of the examples and explanations are directly inspired by or just ripped out of Bjarnes wonderful book "A Tour of C++". I highly recommend that you buy a copy to yourself and all your colleages. Make sure they read it carefully (it is a thin book, there is no excuse) so that you all share at least a basic understanding of modern C++. It will enable you to make educated decisions about what features to use and what to not use.

There are few original ideas of mine in this lecture. The teaching style is inspired by David Beazley. From a C++ knowledge point of view, I am in debt to authors of the many blogs and great books on C++, and to supergurus and my friends Jon Jagger, Lars Gullik Bjønnes and Kevlin Henney. I also learn constantly from my exceptional colleagues at Cisco and from all the smart students attending this and similar courses.

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- rvalues and move semantics
- how to write and understand templates
- function objects and lambda expressions
- decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

Part 1

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
}
```

Instructor notes:

Canonical Hello World program

"Eat, Grandma!" vs "Eat Grandma!"

```
$ c++ tour.cpp && ./a.out
Hello, world!
```

```cpp
#include <iostream>

namespace {

int the_answer()
{
    return 7 * 6;
}

void print_num(int num)
{
    std::cout << num << std::endl;
}

}

int main()
{
    int a = the_answer();
    print_num(a);
}
```

Instructor notes:

discuss **command query separation**

show **namespaces** and **unnamed namespaces**, show **use of static**

```
$ c++ tour.cpp && ./a.out
42
```

```cpp
#include <iostream>

auto the_answer()
{
    return "XLII";
}

template <typename T>
void print_num(T num)
{
    std::cout << num << std::endl;
}

int main()
{
    auto a = the_answer();
    print_num(a);
}
```

Instructor notes:

show **decltype**

show **auto**

discuss auto as function argument, show template version instead

demonstrate return 3.14, "fortytwo" and "XLII"

discuss **generic programming** and **type independent code**

```
$ c++ tour.cpp && ./a.out
XLII
```

```cpp
#include <iostream>

template <typename T>
void print_num(T num)
{
    std::cout << num << std::endl;
}

//void print_num(int num)
//{
//    std::cout << num << std::endl;
//}
//
//void print_num(double num)
//{
//    std::cout << num << std::endl;
//}
//
//void print_num(const char * num)
//{
//    std::cout << num << std::endl;
//}

int main()
{
    print_num(42);
    print_num(3.14);
    print_num("XLII");
}
```

Instructor notes:

discuss what **templates** really are

when a template is not instanciated, no code is generated

```
$ c++ tour.cpp && ./a.out
42
3.14
XLII
$ nm a.out | grep print_num
00000001000011d0 T __Z9print_numIPKcEvT_
0000000100001190 T __Z9print_numIdEvT_
0000000100001150 T __Z9print_numIiEvT_
$
```

```cpp
#include <iostream>

struct Complex {
    double re, im;
};

template <typename T>
void print_num(T num)
{
    std::cout << num << std::endl;
}

template <>
void print_num(Complex c)
{
    std::cout << '(' << c.re << ',' << c.im << ')' << std::endl;
}


int main()
{
    print_num(42);
    print_num(3.14);
    print_num("XLII");
    Complex c{3, -2};
    print_num(c);
}
```

Instructor notes:

Introduce a **user-defined type**

Demonstrate **function overload**

Demonstrate **template specialization**

```
$ output
```

```cpp
#include <iostream>

class Complex {
public:
    explicit Complex(double r, double i) : re(r), im(i) {}
    explicit Complex(double r) : Complex(r, 0) {}
    explicit operator double() { return re; }
    double real() const { return re; }
    double imag() const { return im; }
private:
    const double re, im;
};

std::ostream & operator<<(std::ostream & out, const Complex & c) {
    return out << '(' << c.real() << ',' << c.imag() << ')';
}

template <typename T>
void print_num(const T & num)
{
    std::cout << num << std::endl;
}

int main()
{
    print_num(42);
    print_num(3.14);
    print_num("XLII");
    Complex c{3, -2};
    print_num(c);
}
```

Instructor notes:

discuss the need for **explicit**

discuss **accessors** and **modifiers**, getters/setters

discuss **imutable objecs** vs **mutable objects**

value objects

demonstrate **operator<<** as a **friend**

discuss **value semantics**

show **default values** for members

discuss use of **const**

discuss **initializer list** vs **assignment** in ctor

```
42
3.14
XLII
(3,-2)
```

```cpp
#include <iostream>

template <typename Real, typename Imag>
class Complex {
public:
    explicit Complex(Real r, Imag i) : re(r), im(i) {}
    explicit Complex(Real r) : Complex(r, 0) {}
    explicit operator Real() { return re; }
    Real real() const { return re; }
    Imag imag() const { return im; }
private:
    const Real re;
    const Imag im;
};

template <typename Real, typename Imag>
std::ostream & operator<<(std::ostream & out, const Complex<Real, Imag> & c) {
    return out << '(' << c.real() << ',' << c.imag() << ')';
}

template <typename T>
void print_num(const T & num)
{
    std::cout << num << std::endl;
}

int main()
{
    print_num(42);
    print_num(3.14);
    print_num("XLII");
    Complex<int, double> c{3, -2};
    print_num(c);
}
```

Instructor notes:

demonstrate that Complex and operator<< can/should also be templates

then simplify

```
42
3.14
XLII
(3,-2)
```

```cpp
#include <iostream>

class Complex {
public:
    explicit Complex(double r, double i);
    double real() const;
    double imag() const;
private:
    const double re;
    const double im;
};

Complex::Complex(double r, double i) : re(r), im(i) {}
double Complex::real() const { return re; }
double Complex::imag() const { return im; }

std::ostream & operator<<(std::ostream & out, const Complex & c) {
    return out << '(' << c.real() << ',' << c.imag() << ')';
}

template <typename T>
void print_num(const T & num)
{
    std::cout << num << std::endl;
}

int main()
{
    print_num(42);
    print_num(3.14);
    print_num("XLII");
    Complex c{3, -2};
    print_num(c);
}
```

Instructor notes:

discuss **declaration** vs **definition**

discuss **namespace**

prepare for moving into mylib.hpp and mylib.cpp

```
$ output
```

mylib.hpp

```
#include <iostream>

class Complex {
public:
    explicit Complex(double r, double i);
    double real() const;
    double imag() const;
private:
    const double re;
    const double im;
};

inline std::ostream & operator<<(std::ostream & out, const Complex & c)
{
    return out << '(' << c.real() << ',' << c.imag() << ')';
}
```

Instructor notes:

Step 0:

Separate compilation units (aka **translation units**)

discuss **header guards** (compile without **inline**)

discuss rules for **client code** and **“library” code**

discuss and add **namespaces**

mylib.cpp

```
#include "mylib.hpp"

Complex::Complex(double r, double i) : re(r), im(i)
{
}

double Complex::real() const
{
    return re;
}

double Complex::imag() const
{
    return im;
}
```

tour.cpp

```
#include "mylib.hpp"

template <typename T>
void print_num(T num)
{
    std::cout << num << std::endl;
}

int main()
{
    print_num(42);
    print_num(3.14);
    auto c = Complex{42, -7};
    print_num(c);
}
```

```
$ c++ -c -o mylib.o mylib.cpp
$ c++ -c -o tour.o tour.cpp
$ c++ -o tour tour.o mylib.o
$ ./tour
42
3.14
(42,-7)
$
```

mylib.hpp

```
#ifndef MYLIB_HPP_INCLUDED
#define MYLIB_HPP_INCLUDED

#include <iosfwd>

namespace mylib {

class Complex {
public:
    explicit Complex(double r, double i);
    double real() const;
    double imag() const;
private:
    const double re;
    const double im;
};

}

std::ostream & operator<<(std::ostream & out, const mylib::Complex & c);

#endif
```

Instructor notes:

Dicsuss:
- inside or outside namespace, ADL (Koenig lookup)
- forward declarations, eg <iosfwd>
- namespace pollution
- high cohesion, low coupling
- design vs architectural decisions

Show “using namespace mylib;” in tour.cpp
Discuss importing namespaces in header files, vs implementation files
Discuss implicit vs explicit

mylib.cpp

```
#include "mylib.hpp"
#include <ostream>

namespace mylib {

Complex::Complex(double r, double i) : re(r), im(i)
{
}

double Complex::real() const
{
    return re;
}

double Complex::imag() const
{
    return im;
}

}

std::ostream & operator<<(std::ostream & out, const mylib::Complex & c)
{
    return out << '(' << c.real() << ',' << c.imag() << ')';
}
```

tour.cpp

```
#include "mylib.hpp"
#include <iostream>

using namespace
template <typename T>
void print_num(T num)
{
    std::cout << num << std::endl;
}

int main()
{
    print_num(42);
    print_num(3.14);
    auto c = mylib::Complex{42, -7};
    print_num(c);
}
```

```
$ c++ -c -o mylib.o mylib.cpp
$ c++ -c -o tour.o tour.cpp
$ c++ -o tour tour.o mylib.o
$ ./tour
42
3.14
(42,-7)
$
```

```cpp
#include "mylib.hpp"
#include <iostream>

using mylib::Complex;

Complex operator*(const Complex & a, const Complex & b)
{
    return Complex(a.real() * b.real() - a.imag() * b.imag(),
                   a.real() * b.imag() + a.imag() * b.real());
}

template <typename T>
void print_square(T num)
{
    std::cout << (num * num) << std::endl;
}

int main()
{
    print_square(42);
    print_square(3.14);
    auto c = Complex{42, -7};
    print_square(c);
}
```

Instructor notes:

short on time? maybe skip this slide and the next one

change from print_num to print_square

demo compilation failure

`using namespace mylib`

implement operator*

`using C = mylib::Complex`

`using mylib::Complex`

discuss .hpp vs .cpp issues

move operator* to headerfile

demonstrate link failure

show inline keyword

discus inlining

```
1764
9.8596
(1715,-588)
```

mylib.hpp

```
#ifndef MYLIB_HPP_INCLUDED
#define MYLIB_HPP_INCLUDED

#include <iosfwd>

namespace mylib {

class Complex {
public:
    explicit Complex(double r, double i) : re(r), im(i) {};
    double real() const { return re; };
    double imag() const { return im; };
private:
    const double re;
    const double im;
};

}

inline mylib::Complex operator*(const mylib::Complex & a, const mylib::Complex & b)
{
    return mylib::Complex(
        a.real() * b.real() - a.imag() * b.imag(),
        a.real() * b.imag() + a.imag() * b.real());
}

std::ostream & operator<<(std::ostream & out, const mylib::Complex & c);

#endif
```

Instructor notes:

Dicsuss:
- argument dependant lookup
- optimization
- implementing freestanding functions

tour.cpp

```
#include "mylib.hpp"
#include <iostream>

template <typename T>
void print_square(T num)
{
    std::cout << (num * num) << std::endl;
}

int main()
{
    print_square(42);
    print_square(3.14);
    auto c = mylib::Complex{42, -7};
    print_square(c);
}
```

```
$ c++ -c -o mylib.o mylib.cpp
$ c++ -c -o tour.o tour.cpp
$ c++ -o tour tour.o mylib.o
$ ./tour
42
3.14
(42,-7)
$
```

mylib.cpp

```
#include "mylib.hpp"
#include <ostream>

std::ostream & operator<<(std::ostream & out, const mylib::Complex & c)
{
    return out << '(' << c.real() << ',' << c.imag() << ')';
}
```

Part 2

```cpp
#include <iostream>

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(int * log, std::size_t len)
{
    for (std::size_t i = 0; i < len; ++i)
        transmit_item(log[i]);
}

int main()
{
    int log[] = {24, 20, 37, 42, 34, 45, 38};
    std::size_t len = sizeof log / sizeof log[0];
    transmit_log(log, len);
}
```

Instructor notes:

naive, but typical, code, traditional C style

point out the use of sizeof v / sizeof v[0] idiom

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

```cpp
#include <iostream>

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(int * log, std::size_t len)
{
    const int * begin = log;
    const int * end = log + len;
    for (const int * at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    int log[] = {24, 20, 37, 42, 34, 45, 38};
    std::size_t len = sizeof log / sizeof log[0];
    transmit_log(log, len);
}
```

Instructor notes:

discuss **iterator idiom**

discuss **iterators**

discuss calculating address of one element after an array

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

```cpp
#include <iostream>

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const int * begin, const int * end)
{
    for (const int * at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    int log[] = {24, 20, 37, 42, 34, 45, 38};
    std::size_t len = std::size(log);
    const int * begin = log;
    const int * end = log + len;
    transmit_log(begin, end);
}
```

Instructor notes:

discuss **std::size** , mention **countof** and **ARRAY_SIZE**

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

```cpp
#include <iostream>

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const int * begin, const int * end)
{
    for (const int * at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    int log[] = {24, 20, 37, 42, 34, 45, 38};
    const int * begin = std::cbegin(log);
    const int * end = std::cend(log);
    transmit_log(begin, end);
}
```

Instructor notes:

mention difference between cbegin/cend and begin/end

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int>::const_iterator begin, std::vector<int>::const_iterator end)
{
    for (std::vector<int>::const_iterator at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    std::vector<int> log {24, 20, 37, 42, 34, 45, 38};
    const std::vector<int>::const_iterator begin = std::cbegin(log);
    const std::vector<int>::const_iterator end = std::cend(log);
    transmit_log(begin, end);
}
```

Instructor notes:

show also log.begin() and log.end()

mention **ranges**

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const std::vector<int> & log)
{
    const std::vector<int>::const_iterator begin = std::cbegin(log);
    const std::vector<int>::const_iterator end = std::cend(log);
    for (std::vector<int>::const_iterator at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    std::vector<int> log {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

notes

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const std::vector<int> & log)
{
    auto begin = std::cbegin(log);
    auto end = std::cend(log);
    for (auto at = begin; at != end; ++at)
        transmit_item(*at);
}

int main()
{
    std::vector<int> log {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

notes

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

notes

```
$ c++ tour.cpp && ./a.out
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

talk about the **STL algorithms**

difficult to argue for algorithms when using only one, the power comes when they are combined

```
24
20
37
42
34
45
38
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

std::sort, show compilation error

change to use reference, discuss issues: not good because unexpected behavior

change to call-by-value
discuss issues:
- potentially many items?
- high performace logging?
- trading?

argue that it would be nice to express the idea of passing ownership

```
20
24
34
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    transmit_log(std::move(log));
}
```

Instructor notes:

```cpp
transmit_log(std::move(log));

void transmit_log(std::vector<int> && log)
```

Most important take-away, don't assume anything about the content of something you have "given away". You can delete it, or perhaps reset it and fill it up with new values.

```
20
24
34
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

bool myfilter(int i)
{
    return i <= 35;
}

void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    auto newend = std::remove_if(std::begin(log), std::end(log), myfilter);
    log.erase(newend, std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38});
}
```

Instructor notes:

`transmit_log({24, 20, 37, 42, 34, 45, 38});`

discuss temp object and life-time issues

`bool myfilter(int i)`

`std::remove_if`

`log.erase`

mention the erase-remove idiom, maybe show one-line version

Point out how use of cbegin()/cend() vs begin()/end() communicates well

However, this is probably not very efficient… what about….

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

bool myfilter(int i)
{
    return i <= 35;
}

void transmit_log(std::vector<int> && log)
{
    auto newend = std::remove_if(std::begin(log), std::end(log), myfilter);
    std::sort(std::begin(log), newend);
    std::for_each(std::begin(log), newend, transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38});
}
```

Instructor notes:

move remove_if first

```cpp
std::sort(std::begin(log), newend);

std::for_each(std::begin(log), newend, transmit_item);
```

Discuss why const iter and mutable iter can’t be combined in for_each

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

int myfilter_limit = 35;
bool myfilter(int i)
{
    return i <= myfilter_limit;
}

void transmit_log(std::vector<int> && log, int limit)
{
    myfilter_limit = limit;
    auto newend = std::remove_if(std::begin(log), std::end(log), myfilter);
    std::sort(std::begin(log), newend);
    std::for_each(std::begin(log), newend, transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38}, 35);
}
```

Instructor notes:

parameterize from above

```cpp
void transmit_log(std::vector<int> && log, int limit)

transmit_log({24, 20, 37, 42, 34, 45, 38}, 35);
```

... add a question mark first line in transmit_log. How to solve this?

Warn upfront about the following terrible (but unfortunately, quite common) solution:

```cpp
int myfilter_limit = 35;

return i < myfilter_limit;

myfilter_limit = limit;
```

Experiment: ask for suggestions for a better solution

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

struct filter {
    filter(int limit) : lim(limit) {}
    bool operator()(int i) {
        return i <= lim;
    }
    int lim;
};

void transmit_log(std::vector<int> && log, int limit)
{
    filter myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38}, 35);
}
```

Instructor notes:

for simplicity, move back to the erase-remove idiom

use cbegin()/cend() in for_each

comment out the myfilter predicate and the first line in transmit_log

implement struct filter

`filter myfilter(limit);`

introduce the function object idea (aka functor).

Point out that remove_if does not care what the filter is as long as it can be called

While it has “always” been possible to do this in C++, it is not often seen in user code

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) {
            return i <= lim;
        }
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38}, 35);
}
```

Instructor notes:

Inline the filter

It would be nice to have an unamed functor here… but due to ctor…

Maybe show that _ as a name works… not don't do that… yes do it… arghh… :-)

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> && log, int limit)
{
    //struct filter {
    //    filter(int limit) : lim(limit) {}
    //    bool operator()(int i) {
    //        return i <= lim;
    //    }
    //    int lim;
    //} myfilter(limit);
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38}, 35);
}
```

Instructor notes:

comment out the functor

add the lambda expression / closure

point out the similarities between a lambda and a functor, indeed the lambda expression creates a function object, and some compilers (I believe) actually implements lambda by creating an old-school functor.

point out that the lambda like an unamed function with state

we have a named object, but an unamed type

discuss capture, by reference, by value, mutable, all or some

mentions the potential havoc you can do with big and long lived lambdas

```
37
38
42
45
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    auto myfilter = [limit=35](int i) { return i <= limit; };
    transmit_log({24, 20, 37, 42, 34, 45, 38}, myfilter);
}
```

Instructor notes:

first expand auto into std::function<bool (int)>, then make an argument

move the lambda expression into main, assign limit to a value

Generalized lambda captures (C++14)

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    transmit_log({24, 20, 37, 42, 34, 45, 38}, [](int i) { return i <= 35; });
}
```

Instructor notes:

just inline the lambda and simplify it

```
37
38
42
45
```

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

template <typename LogItem>
void transmit_item(LogItem i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filter>
void transmit_log(Log && log, Filter myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item<typename Log::value_type>);
}

int main()
{
    using item_type = int;
    transmit_log(std::vector<item_type>{24, 20, 37, 42, 34, 45, 38}, [](int i) { return i <= 35; });
}
```

Instructor notes:

template <typename Log>

show compilation error, type deduction not possible

add std::vector<int> in main

template <typename LogItem>

show comilation error, type deduction failed

add first

transmit_item<int>

then

transmit_item<typename Log::value_type>

show error message

transmit_item<typename Log::value_type>

typename is needed because Log::value_type is a dependent name, and we need to specify that we are talking about a type and not a value of some type.

template <typename Log, typename Filter>
void transmit_log(Log && log, Filter myfilter)

show that you can change item_type to double
show that you can use std::deque<item_type> instead
discuss the pros and cons of generic and type independent code

```
37
38
42
45
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>

template <typename LogItem>
void transmit_item(LogItem i)
{
    static_assert(std::is_integral<LogItem>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filter>
void transmit_log(Log && log, Filter myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item<typename Log::value_type>);
}

int main()
{
    using item_type = short;
    transmit_log(std::vector<item_type>{24, 20, 37, 42, 34, 45, 38}, [](int i) { return i <= 35; });
}
```

Instructor notes:

#include <type_traits>

static_assert

type traits is a way to reason about the type of things in compile time

using item_type = double;

show error message

using item_type = short;

```
37
38
42
45
```

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

template <typename T>
struct is_transmittable {
    static bool const value = false;
};

template <>
struct is_transmittable<short> {
    static bool const value = true;
};

template <typename LogItem>
void transmit_item(LogItem i)
{
    static_assert(is_transmittable<LogItem>::value, "transmittable type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filter>
void transmit_log(Log && log, Filter myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::cbegin(log), std::cend(log), transmit_item<typename Log::value_type>);
}

int main()
{
    using item_type = short;
    transmit_log(std::vector<item_type>{24, 20, 37, 42, 34, 45, 38}, [](int i) { return i <= 35; });
}
```

Instructor notes:

remove <type_traits> and <functional>

write both templates

update the **static_assert**

show that it works by changing item_type to double and then int

point out the **explicit (full) template specialization**. Mention **partial template specialization**.

mention **SFNIAE**

mention **concepts** but don't demonstrate it

```
37
38
42
45
```

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

void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log);
}
```

Instructor notes:

clean up

prepare for discussion about concurrency

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>
#include <future>

void transmit_item(int i)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    // ...
    std::cout << i << std::endl;
}

std::promise<std::size_t> nitems_promise;

void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
    nitems_promise.set_value(log.size());
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    auto t = std::thread(transmit_log, log);
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(57));
        std::cout << "Do something else..." << std::endl;
    }
    size_t nitems = nitems_promise.get_future().get();
    std::cout << "# " << nitems << std::endl;
    t.join();
}
```

Instructor notes:

```cpp
std::this_thread::sleep_for(std::chrono::milliseconds(100));

#include <thread>
#include <chrono>
```

Point out that the call to transmit_log is blocking, it would be nice to do something else meanwhile…

```cpp
    auto t = std::thread(transmit_log, log);
    t.join();
```

But how to return data from transmit_log?

```cpp
std::promise<std::size_t> nitems_promise;
nitems_promise.set_value(log.size());
size_t nitems = nitems_promise.get_future().get();
```

Then “edge in” the for loop

```
c++ tour.cpp && ./a.out
Do something else...
24
Do something else...
Do something else...
20
Do something else...
Do something else...
37
Do something else...
Do something else...
42
Do something else...
34
Do something else...
Do something else...
45
38
# 7
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>
#include <future>

void transmit_item(int i)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    // ...
    std::cout << i << std::endl;
}

std::size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(57));
        std::cout << "Do something else..." << std::endl;
    }
    std::size_t nitems = res.get();
    std::cout << "# " << nitems << std::endl;
}
```

Instructor notes:

```
c++ tour.cpp && ./a.out
Do something else...
24
Do something else...
Do something else...
20
Do something else...
Do something else...
37
Do something else...
42
Do something else...
Do something else...
34
Do something else...
Do something else...
45
38
# 7
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>
#include <future>
#include <mutex>

using namespace std::chrono_literals;

std::mutex transmit_mutex;

void transmit_item(int i)
{
    transmit_mutex.lock();
    std::this_thread::sleep_for(100ms);
    // ...
    std::cout << i << std::endl;
    transmit_mutex.unlock();
}

size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(57ms);
        transmit_item(9999);
    }
    std::size_t nitems = res.get();
    std::cout << "# " << nitems << std::endl;
}
```

Instructor notes:

add this after includes

using namespace std::chrono_literals;

show this can also fit in right before usage (actually quite common).
Notice the new scope.

Mention user-defined literals.

change to:

transmit_item(9999);

comment out the sleeps and run a few times

#include <mutex>
std::mutex transmit_mutex;
transmit_mutex.lock();
transmit_mutex.unlock();

Discuss the problem with this acquire / release pattern, especially in languages with exceptions.

```
c++ tour.cpp && ./a.out
24
9999
20
9999
37
9999
42
9999
34
9999
45
9999
38
9999
9999
9999
9999
# 7
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>
#include <chrono>
#include <future>
#include <mutex>

using namespace std::chrono_literals;

std::mutex transmit_mutex;

struct mylock {
    explicit mylock(std::mutex & m): mutex_(m) { m.lock(); }
    ~mylock() { mutex_.unlock(); }
    std::mutex & mutex_;
};

void transmit_item(int i)
{
    mylock lock(transmit_mutex);
    std::this_thread::sleep_for(100ms);
    // ...
    std::cout << i << std::endl;
}

size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log = {24, 20, 37, 42, 34, 45, 38};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(57ms);
        transmit_item(9999);
    }
    std::size_t nitems = res.get();
    std::cout << "# " << nitems << std::endl;
}
```

Instructor notes:

implement mylock with the **RAII idiom**

mention smart pointers instead of new/delete

can be used for open/close files and connections

...

```
output 18
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <future>
#include <mutex>
#include <numeric>

long total_transmitted_items(0);

std::mutex transmit_mutex;

void transmit_item(int i)
{
    std::lock_guard<std::mutex> guard(transmit_mutex);
    (void)i;
    // ...
    total_transmitted_items++;
}

void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log(5000);
    std::iota(std::begin(log), std::end(log), 10000);
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i = 0; i < 5000; ++i)
        transmit_item(99999);
    std::cout << total_transmitted_items << std::endl;
}
```

Instructor notes:

notes

```
output 18
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <future>
#include <mutex>
#include <numeric>

std::atomic<long> total_transmitted_items(0);

void transmit_item(int i)
{
    (void)i;
    // ...
    total_transmitted_items++;
}

void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::cbegin(log), std::cend(log), transmit_item);
}

int main()
{
    std::vector<int> log(5000);
    std::iota(std::begin(log), std::end(log), 10000);
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i = 0; i < 5000; ++i)
        transmit_item(99999);
    std::cout << total_transmitted_items << std::endl;
}
```

Instructor notes:

first remove the lock_guard

demonstrate what happens

add the std::atomic<long>

```
10000
```

```c
#include <stdio.h>

void transmit_item(int i)
{
    printf("%d\n", i);
}

void transmit_log(const int * log, size_t len)
{
    for (size_t i = 0; i < len; i++)
        transmit_item(log[i]);
}

int main(void)
{
    int log[] = {24, 20, 37, 42, 34, 45, 38};
    transmit_log(log, sizeof log / sizeof log[0]);
}
```

Instructor notes:

It is worth pointing out that most C programs are also perfectly valid C++ programs. So if you have no special needs, and don't want to use all the features of modern C++ (yet), then you can of course just write your program like this…

```
24
20
37
42
34
45
38
```

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- rvalues and move semantics
- how to write and understand templates
- function objects and lambda expressions
- decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- rvalues and move semantics
- how to write and understand templates
- function objects and lambda expressions
- decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- function objects and lambda expressions
- decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- ✓ iterators, smart pointers and object lifetimes
- using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- ✓ iterators, smart pointers and object lifetimes
- ✓ using the standard library effectively
- misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- ✓ iterators, smart pointers and object lifetimes
- ✓ using the standard library effectively
- ✓ misc do's and don'ts in modern C++
- modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- ✓ iterators, smart pointers and object lifetimes
- ✓ using the standard library effectively
- ✓ misc do's and don'ts in modern C++
- ✓ modern design principles and how to write solid code
- C++11/14/17 and where C++ is heading

A Tour of Modern C++ (the 2.5 hour version)

In this session I hope you will learn at least some small hints about:

- ✓ rvalues and move semantics
- ✓ how to write and understand templates
- ✓ function objects and lambda expressions
- ✓ decltype, auto and type deduction in general
- ✓ exception handling and exception safety
- ✓ "mystical" stuff like ADL, RAII and SFINAE
- ✓ futures, promises and higher-order parallelism
- ✓ concepts and type traits
- ✓ iterators, smart pointers and object lifetimes
- ✓ using the standard library effectively
- ✓ misc do's and don'ts in modern C++
- ✓ modern design principles and how to write solid code
- ✓ C++11/14/17 and where C++ is heading

!