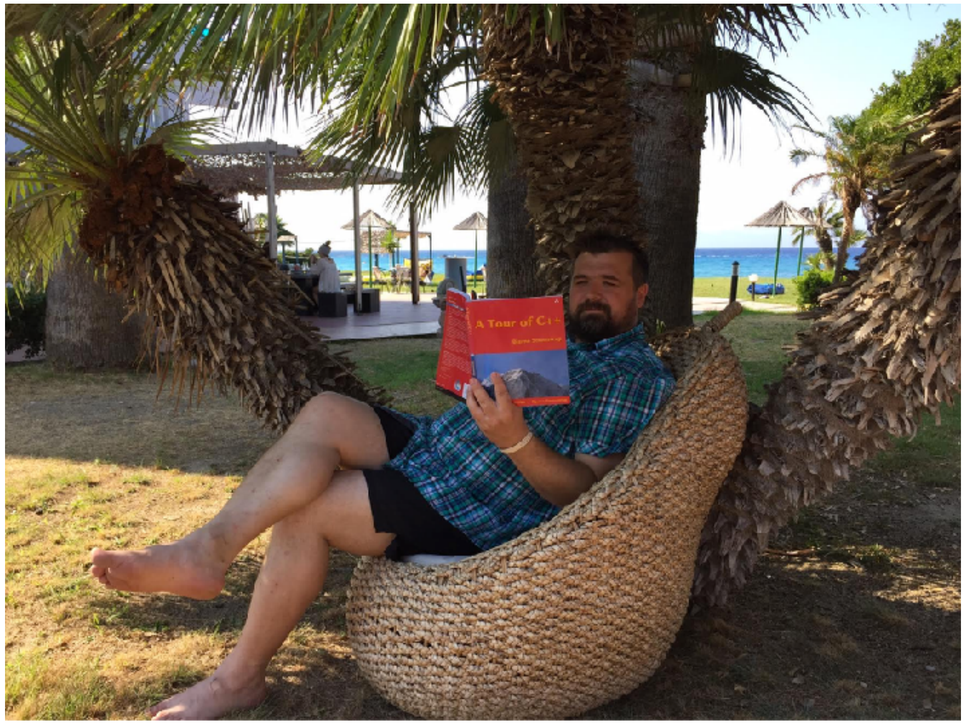
A Tour of Modern C++

with Olve Maudal



Instructor notes

8 hour live coding lecture at

NDC TechTown, Kongsberg, October 18, 2017

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.

Workshop: A Tour of Modern C++

C++

In this fast-paced course we will start from scratch and relearn C++ with modern syntax and semantics. Among other things you will learn (at least something) 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, C++14 and new stuff coming with C++17 and later

This course is aimed at experienced programmers that would like to learn how to write, or at least understand, modern C++. Ideally you should have some experience with either C, old-school C++, Python and/or Java.

Computer setup

Students will need to bring their own laptops with a C++ compiler and a recent C++ development environment.

Speaker

Olve Maudal

Date

18 Oct

Start time

09:00

End time

17:00

A Tour of Modern C++

In this fast-paced course we will start from scratch and relearn C++ with modern syntax and semantics. Among other things you will learn (at least something) 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, C++14 and new stuff coming with C++17 and later

This course is aimed at experienced programmers that would like to learn how to write, or at least understand, modern C++. Ideally you should have some experience with either C, old-school C++, Python and/or Java.

Part 1

(type deduction, templates, user-defined types)

```
int main(){}
```

Instructor notes:

Minimal C++ program (13 chars, because a valid C++ must have a final newline as well)

Aside:

This program is indeed potentially useful. Since it is guaranteed to return 0 (EXIT_SUCCESS) to the runtime environment, it would be a perfectly valid C++ implementation of the Unix /bin/true program

```
$ c++ tour.cpp
$ ./a.out
$ echo $?
0
$
```

```
int main()
{
    return 1;
}
```

Instructor notes:

`main()` is the `entry point` for all C++ programs

there are certain special rules for `main()`

You can of course return a specific value if you want.

```
$ c++ -o myfalse tour.cpp
$ ./myfalse
$ echo $?
1
$ /usr/bin/false
$ echo $?
1
$
```

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

Instructor notes:

The canonical **“Hello, World!”** program in C++

(Don’t forget the comma: “Eat, grandma!” vs “Eat grandma!”)

<< is just an operator taking two operands. This is the infix form

```
#include <iostream>

int main()
{
    operator<<(std::cout, "Hello, World!\n");
}
```

Instructor notes:

While unusual, you can also write:
`operator<<(std::cout, “Hello, World!\n”)`

It is sometimes very useful to mentally convert an expression into it’s prefix equivalence.

```cpp
#include <iostream>

int main()
{
    int d;
    d = 4.8 / 2 + 8 * 5;
    std::cout << d << std::endl;
}
```

Instructor notes:

(probably skip this step)

a declaration is a statement that introduces a name into the program.
a definition is a unique specification of an object, function, class or enumerator.

type, object, value, variable (see page 5-6)
a type defines a set of possible values and operations
an object is some memory that holds a value of some type a value is a set of bits intepreted according to a type
a variable is a named object

Storage duration:
objects of automatic storage duration (local variables) objects of static storage duration (global variables) objects of dynamic storage duration (allocated variables) objects of thread storage duration (thread variables)

Scope:
local scope
class scope
namespace scope (global name = global namespace)

Linkage:
no linkage (only visible to the scope it is in)
internal linkage (visible to the translation unit)
external linkage (can be referred to from other translation units)

```cpp
#include <iostream>

int main()
{
    int d;
    double a = 4.8;
    long b(8);
    int c{5};

    d = a / 2 + b * c;

    std::cout << d << std::endl;
}
```

```
$ c++ -o theanswer tour.cpp
$ ./theanswer
42
$
```

Instructor notes:

a C++ program concists of **expressions** and **statements**

`int d;` is a **statement** that creates a named object (aka variable) of type `int` but at this point the object will have an **indeterminate value**, and if you try to read the value of an object with indeterminate value you get **undefined behavior**

`double a = 2.2;` creates a named object of type `double` that is immediately initialized with a value.

`long b(8);` will also be initialized to a value

`int c{5};` is the modern way to initialize objects in C++, so called **uniform initialization** (aka **braced initialization**)

d = a + b * c; is a statement that evaluates an expression. It is an **expression statement**. Here we have 3 **operators** and 4 **operands**. Unlike many other languages, there is **no *assignment statement*** in C++, instead assignment happens as a **sideeffect** during evaluation involving the **assignment operator**. Indeed, the computed value after evaluating an expression is discarded.

Also, unlike many other languages, the **evaluation order** is mostly **unspecified** in C++. The compiler is free to evaluate `b * c` before evaluating `a / 2` but here it does not matter for the end result. Ie, while many languages uses a l**eft-to-right evaluation** of expressions, C++ does not. This can have dramatic and catastophic effect if the evaluating **subexpressions** have sideeffects.

`std::cout << d << std::endl;` is also an expression statement with a useful sideffect. << , the left shift operator has been overloaded so if given the right operands it can have the sideeffect of writing stuff to the **standard console output stream** (aka stdout). However, here the evaluation order is guaranteed to be left-to-right.

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

std::string operator+(std::string str, int i)
{
    return str + std::to_string(i);
}

int main()
{
    int d;
    double a = 4.8;
    long b(8);
    int c{5};

    // int = double / int + long * int
    // int = double / int + long * long(int)
    // int = double / int + long(long * long(int))
    // int = double / double(int) + long(long * long(int))
    // int = double(double / double(int)) + long(long * long(int))
    // int = double(double / double(int)) + double(long(long * long(int)))
    // int = double(double(double / double(int)) + double(long(long * long(int))))
    // int = int(double(double(double / double(int)) + double(long(long * long(int)))))
    d = a / 2 + b * c;

    // ostream << int << iomanip
    // operator<<(ostream, int) << iomanip
    // ostream << iomanip
    // operator<<(ostream, iomanip)
    // ostream
    std::cout << d << std::endl;

    std::string s{"Level"};
    // ostream << (std::string + int) << iomanip
    // ostream << (operator+(std::string, int)) << iomanip
    // ostream << std::string << iomanip
    // operator<<(ostream, std::string) << iomanip
    // ostream << iomanip
    // operator<<(ostream, iomanip)
    // ostream
    std::cout << (s + d) << std::endl;
}
```

Instructor notes:

When reasoning about C++ you often must focus on the **types** of objects and expressions, otherwise the code does not make sense.

The operators in C++ are usually very specific on what the types of its operands can be. The compiler needs to figure out a way to **convert** the objects to certain types (and sometimes changing the value in the process, eg during **narrowing**, but also for example when promoting an `int` to a `double`) before they can worked on by the operator.

These conversions are often silent and happens **implicitly**. The conversion rules in C++ are very complicated.

C++ is a language defined by its behavior (not implementation)

In this case, the actual machine code produced by the compiler can be whatever as long as it eventually prints 42 and then Level42 to the standard console output stream. If you turn on optimization it very likely that the compiler will just produce code equivalent to:

```cpp
#include <stdio.h>

int main()
{
  puts("42");
  puts("Level42");
}
```

Experiment: Try to `c` to `unsigned int` and change to `b(-8)`, and then compile with `-m32`

```
42
Level42
```

```cpp
#include <iostream>

int the_answer()
{
    return 4.8 / 2 + 8 * 5;
}

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

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

Instructor notes:

Talk about **command query separation**

Experiment: try `char * a = the_answer()` - observe the error message. The compiler knows the type of `the_answer()`. Now show:

```cpp
decltype(the_answer) a = the_answer();

auto = the_answer();

auto the_answer()

auto the_answer() -> int
```

```
42
```

```cpp
#include <iostream>

auto the_answer()
{
    return 4.8 / 2 + 8 * 5;
}

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

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

Instructor notes:

Type deduction is not new, it is the key thing that makes template work

```
void print_num(auto num) // -fconcepts
```

Aside:

```
#include <typeinfo>
std::cout << typeid(a).name() << std::endl;
```

```
42
```

```cpp
#include <iostream>
#include <typeinfo>

auto the_answer()
{
    return 4.8 / 2 + 8 * 5;
}

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

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

    std::cout << typeid(the_answer()).name() << std::endl;
    std::cout << typeid(print_num(a)).name() << std::endl;
    std::cout << typeid(a).name() << std::endl;
}
```

Instructor notes:

discuss generic programming and type independent code

change to return 6 * 7 and then “42”

C++ is moving in the direction of generic programming (one reason why learning, say, Python is a good way to understand C++ properly)

```
42.4
d
v
d
```

```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;
//}

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

Instructor notes:

focus on **templates**

This code will create two different `print_num` functions, it is exactly as if…
(expand into two f**unction overloads**)

here the template is used two times to “stamp out” two functions

when template is not used, no code is created

Experiment: try this with and without calling `print_num`:

```
c++ tour.cpp
nm ./a.out
```

```
42
3.14
```

```cpp
#include <iostream>

struct Complex {
    double re, im;
};

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

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

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

Instructor notes:

Introducing a **user-defined type**, simple struct Complex

Here we do **pass by value**, ie the object will be copied

Demonstrate:
- **pass by reference**
- change to: `const Complex c;` discuss the error message
- pass by **const reference**
- discuss **temporary objects**
- maybe also show “pass by pointer”

```cpp
print_num(Complex{3, -2});

print_num({3, -2});

template <>

void print_num(Complex c)

template <typename T=Complex>

void print_num(const Complex & c)
```

Discuss programming and communication

discuss **value samentics**
references as “first class citizens”
demonstrate simple change from const reference back to value

show and discuss **template specialization**

Aside: Complex like this is a **POD** (Plain Old Data), guaranteed to behave like a plain struct in C

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

```cpp
#include <iostream>

struct Complex {
    double re, im;
};

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

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

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

Instructor notes:

implement **operator overloading** for our user-defined type

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

```cpp
#include <iostream>

struct Complex {
    explicit Complex(double r, double i) : re(r), im(i) {}
    explicit Complex(double r) : Complex(r,0) {}
    explicit operator double() { return re; }
    double re, im;
};

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

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

int main()
{
    print_num(42);
    print_num(3.14);
    Complex c{3, -2};
    print_num(c);
    double v = double(c);
    print_num(v);
}
```

Instructor notes:

first write the two constructors **without the explicit specifier** (don't implement the conversion operator yet)

change template to `print_num(Complex num)` to demonstrate **silent implicit conversion** to Complex

add explicit to both constructors, discuss why

add conversion operator, demonstrate how it works

change template to print_num(int) and demonstrate implicit conversion

add explicit also to the conversion operator

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

```cpp
#include <iostream>

template <typename T>
struct Complex {
    explicit Complex(T r, T i) : re(r), im(i) {}
    explicit Complex(T r) : Complex(r,0) {}
    explicit operator T() { return re; }
    T re, im;
};

template <typename T>
std::ostream & operator<<(std::ostream & out, const Complex<T> & c)
{
    return out << '(' << c.re << ',' << c.im << ')';
}

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

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

Instructor notes:

first change struct Complex to use int instead and motivate for using a class template

implement class template, show compilation error on operator<<

make operator<< a template as well

discuss type deduction for templates

Note: In number theory, a Gaussian integer is a complex number whose real and imaginary parts are both integers.

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

```cpp
#include <iostream>

template <typename Real, typename Imag>
struct Complex {
    explicit Complex(Real r, Imag i) : re(r), im(i) {}
    explicit Complex(Real r) : Complex(r,0) {}
    explicit operator Real() { return re; }
    Real re;
    Imag im;
};

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

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

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

Instructor notes:

template with multiple arguments

discuss use and overuse of templates

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

```cpp
#include <iostream>

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

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

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

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

Instructor notes:

clean up before discussing class vs struct

```
42
3.14
(42,-7)
```

```cpp
#include <iostream>

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

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

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

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

Instructor notes:

just add **access specifiers** first (`public` and `private`), show that `operator<<` now does not work. Show how **friend** can be used, but discourage its use.

add **accessors** (aka “getters”)

show what happens if we remove the **const qualifier** after accessors. Discuss the “guarantee” that `operator<<` has given to the caller. Try changing “`const Complex &`” to “`Complex c`”

change to `class Complex`

```
42
3.14
(42,-7)
```

```
#include <iostream>

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

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

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.imag(-4);
    print_num(c);
}
```

Instructor notes:

add **modifiers** (aka “setters”)

add **default values** to ctor

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

value objects

discuss the need for default values… do we really need them?

do you really need “setters”?

show **default values** for members

```
output
```

```cpp
#include <iostream>

struct 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;
};

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

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);
}
```

Instructor notes:

clean up, remove modifiers

discuss the **initializer list** show assignment vs initialization, eg:

```cpp
explicit Complex(double r, double i) : re(), im() {
    re = r;
    im = i;
}
```

show that re and im can (and should) be const

```
42
3.14
(42,-7)
```

```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(T num)
{
    std::cout << num << std::endl;
}

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

Instructor notes:

discuss declaration vs definition

discuss namespace

add unnamed namespace

prepare for moving into mylib.hpp and mylib.cpp

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

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
```

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)
$
```

Instructor notes:

Dicsuss:
- inside or outside namespace, ADL (Koenig lookup)
- forward declarations, eg <iosfwd>
- namespace pollution (again)
- 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() << ')';
}
```

```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:

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 I - Summary

minimal C++ program
Hello, World!
the << operator
infix vs prefix
expressions and statements
operators and operands
conversion rules
widening and narrowing
promotion and demotion
implicit and explicit conversions
indeterminate values
undefined behavior
unspecified behavior
scope of variables
object life-times
type, object, value, variable
fundamental types
assignment and initialization
uniform/braced initialiation
evaluation order
subexpressions
side effects
defined by behavior
optimization
command query separation
type deduction
templates
function templates
class template
template specialization
generic programming
type independent code
C++ and Python
function overload
user-defined type
pass by value
pass by reference
const reference
"pass by pointer"
value semantics
temporary objects
POD - Plain Old Data
operator overload
implicit conversions
explicit specifier
constructor
conversion operator
Gaussian integer
templates with multiple argument
class vs struct
access specifiers
friend
accessors and modifiers
const qualifiers
default values
mutability
imutable value objects
initializer list
declaration and definitions
namespace
unnamed namespace
compilation units
header guards
client vs library code
ADL - argument dependent lookup
forward declarations
namespace pollution
cohesion and coupling
design and architecture
headers and implementation files
using and typedef
inline
freestanding functions
optimization

Part 2

(iterators, move semantics, STL)

```cpp
#include <iostream>
#include <cstddef>

int main()
{
    int v[4] = {2, 3, 5, 7};
    std::size_t n_elems = sizeof v / sizeof v[0];
    for (std::size_t i = 0; i < n_elems; ++i)
        std::cout << (v[i] * v[i]) << '\n';
}
```

Instructor notes:

discuss `int` vs `size_t` vs `std::size_t` vs *`container`*`::size_type`

mention namespace pollution (try uint instead of int)

(do **not** discuss ++i vs i++)

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

int main()
{
    int v[] = {2, 3, 5, 7};
    const int * begin = &v[0];
    const int * end = &v[sizeof v / sizeof v[0]];
    for (const int * iter = begin; iter != end; ++iter)
        std::cout << (*iter * *iter) << '\n';
}
```

Instructor notes:

discuss iterators

discuss calculating address of one element after an array

the use of sizeof v / sizesof v[0] idiom (mention countof)

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

int main()
{
    int v[] = {2, 3, 5, 7};
    const int * begin = std::begin(v);
    const int * end = std::end(v);
    for (const int * iter = begin; iter != end; ++iter)
        std::cout << (*iter * *iter) << '\n';
}
```

Instructor notes:

point out how namespaces allow us to reuse the names begin and end

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

int main()
{
    int v[] = {2, 3, 5, 7};
    auto begin = std::cbegin(v);
    auto end = std::cend(v);
    for (auto iter = begin; iter != end; ++iter)
        std::cout << (*iter * *iter) << '\n';
}
```

Instructor notes:

notes

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

int main()
{
    int v[] = {2, 3, 5, 7};
    for (auto iter = std::cbegin(v); iter != std::cend(v); ++iter)
        std::cout << (*iter * *iter) << '\n';
}
```

Instructor notes:

show range-for version, but revert back to regular for loop, we need it for clarity in the next steps

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

void print_squares(const int * v, std::size_t len)
{
    const int * begin{v + 0};
    const int * end{v + len};
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[] = {2, 3, 5, 7};
    print_squares(v, sizeof v / sizeof v[0]);
}
```

Instructor notes:

gotcha: passing array into a function, eg int v[] as a function argument

array will decay into a pointer… “metainformation” is lost

```
4
9
25
49
```

```cpp
#include <iostream>
#include <cstddef>

void print_squares(const int * begin, const int * end)
{
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[] = {2, 3, 5, 7};
    print_squares(v + 0, v + sizeof v / sizeof v[0]);
}
```

Instructor notes:

instead of passing basepointer and size, we pass the iterator begin and end

```
4
9
25
49
```

```cpp
#include <iostream>

struct Array {
    size_t sz;
    int * data;
};

void print_squares(const int * begin, const int * end)
{
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[] = {2, 3, 5, 7};
    Array a{sizeof v / sizeof v[0], v};
    print_squares(a.data, a.data + a.sz);
}
```

Instructor notes:

Let’s start implementing our own collection type, an Array

```
4
9
25
49
```

```cpp
#include <iostream>

struct Array {
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    const int * begin = a.data;
    const int * end = a.data + a.sz;
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[] = {2, 3, 5, 7};
    Array a{sizeof v / sizeof v[0], v};
    print_squares(a);
}
```

Instructor notes:

notes

```
4
9
25
49
```

```cpp
#include <iostream>

struct Array {
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    const int * begin = a.data;
    const int * end = a.data + a.sz;
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    //int v[] = {2, 3, 5, 7};
    Array a(4);
    a[0] = 2;
    a[1] = 3;
    a[2] = 5;
    a[3] = 7;
    print_squares(a);
}
```

Instructor notes:

implement the constructor and destructor

discuss the order of initialization
discuss the new int[size]() “trick”

show that after ctor/dtor we can no long use C style initialization

implement operator[] so that we can conveniently set elements

```
4
9
25
49
```

```cpp
#include <iostream>

struct Array {
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    const int * begin = a.data;
    const int * end = a.data + a.sz;
    for (const int * at = begin; at != end; ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[] = {2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);
    int * iter = &v[0];
    int * output_iter = &a[0];
    while (iter != &v[n])
        *output_iter++ = *iter++;
    print_squares(a);
}
```

Instructor notes:

notes

```
4
9
25
49
```

```cpp
#include <iostream>

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const int * at = std::begin(a); at != std::end(a); ++at)
        std::cout << (*at * *at) << '\n';
}

int main()
{
    int v[]{2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);

    // copy
    int * iter = std::begin(v);
    int * output_iter = std::begin(a);
    while (iter != std::end(v))
        *output_iter++ = *iter++;

    print_squares(a);
}
```

Instructor notes:

first implement begin() and end()

show that begin and end in print_squares can now use a.begin() and a.end()

show std::begin(v) and go via a.begin() to std::begin(a)

now that we have a way to access the data through iterators and accessor we can add access specifiers

prepare discussion for a copy() function

```
4
9
25
49
```

```
#include <iostream>

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const int * at = std::begin(a); at != std::end(a); ++at)
        std::cout << (*at * *at) << '\n';
}

void copy(const int * iter, const int * end, int * dest)
{
    while (iter != end)
        *dest++ = *iter++;
}

int main()
{
    int v[]{2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);
    copy(std::cbegin(v), std::cend(v), std::begin(a));
    print_squares(a);
}
```

Instructor notes:

discuss how use of cbegin()/cend() communicates well

discuss the generic nature of this copy func

create a template version of void copy(), remember to use copy<int>

```
4
9
25
49
```

```
#include <iostream>

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const int * at = std::begin(a); at != std::end(a); ++at)
        std::cout << (*at * *at) << '\n';
}

template <typename T>
void copy(const T * iter, const T * end, T * dest)
{
    while (iter != end)
        *dest++ = *iter++;
}

int main()
{
    int v[]{2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);
    copy<int>(std::cbegin(v), std::cend(v), std::begin(a));
    print_squares(a);
}
```

Instructor notes:

remove <int> from copy<int>, and discuss type deduction

change to T * copy()

```
4
9
25
49
```

```
#include <iostream>

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

template <typename T>
T * copy(const T * iter, const T * end, T * dest)
{
    while (iter != end)
        *dest++ = *iter++;
    return dest;
}

int main()
{
    int v[]{2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);
    copy(std::cbegin(v), std::cend(v), std::begin(a));
    print_squares(a);
}
```

Instructor notes:

introduce range-for, go via:

for (auto v : a)

introduce std::copy from <algorithm>

```
4
9
25
49
```

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

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    int v[]{2, 3, 5, 7};
    std::size_t n = sizeof v / sizeof v[0];
    Array a(n);
    std::copy(std::cbegin(v), std::cend(v), std::begin(a));
    print_squares(a);
}
```

Instructor notes:

notes

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>

class Array {
public:
    Array(size_t size) : sz(size), data(new int[size]()) {}
    Array(const std::initializer_list<int> & list) :
        sz(list.size()), data(new int[sz]) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    ~Array() { delete[] data; }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
}
```

Instructor notes:

constructor for initializer_list<int>

gotcha: what about Array a{4} vs Array a(4)

make Array(size) private

demonstrate Array a{} vs Array a(); but don’t spend time on vexing parse issues

```
4
9
25
49
```

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

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    //Array(const Array & other) : sz(other.sz), data(other.data) {}
    ~Array() { delete[] data; }
    //Array & operator=(const Array & other) {
    //    sz = other.sz;
    //    data = other.data;
    //    return *this;
    //}
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
    Array b = a;
    print_squares(b);
}
```

Instructor notes:

demonstrate UB and crash Array b = a

demonstrate UB and crash Array b; b = a

discuss construction vs assignment

implement the shallow copy constructor

implement the shallow assignment operator

comment out copy ctro and assignment operator

```
4
9
25
49
4
9
25
49
a.out(25363,0x7fffbae4b3c0) malloc: *** error for object 0x7fd25dc00350:
pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
/bin/bash: line 1: 25363 Abort trap: 6           ./a.out
```

```
#include <iostream>
#include <algorithm>

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) : Array(other.sz) {
        std::copy(std::cbegin(other), std::cend(other), data);
    }
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) {
        delete[] data;
        sz = other.sz;
        data = other.data;
        std::copy(std::cbegin(other), std::cend(other), data);
        return *this;
    }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
    Array b = a;
    print_squares(b);
}
```

Instructor notes:

discuss the old-school “rule of three”

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) : Array(other.sz) {
        std::copy(std::cbegin(other), std::cend(other), data);
    }
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) {
        Array tmp(other);
        std::swap(sz, tmp.sz);
        std::swap(data, tmp.data);
        return *this;
    }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
    Array b = a;
    print_squares(b);
}
```

Instructor notes:

show the swap idiom and discuss “gutting”

discuss exception safety

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) : Array(other.sz) {
        std::copy(std::cbegin(other), std::cend(other), data);
    }
    Array(Array && other) : sz(other.sz), data(other.data) {
        other.sz = 0;
        other.data = nullptr;
    }
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) {
        Array tmp(other);
        std::swap(sz, tmp.sz);
        std::swap(data, tmp.data);
        return *this;
    }
    Array & operator=(Array && other) {
        std::swap(sz, other.sz);
        std::swap(data, other.data);
        return *this;
    }
    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
    Array b{};
    b = std::move(a);
    print_squares(b);
}
```

Instructor notes:

talk about move semantics and rvalue references

mention rule of 5

after std::move, we can't refer to it again, only destroy it

security issues with gutting?

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) = delete;
    Array(Array && other) = delete;
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) = delete;
    Array & operator=(Array && other) = delete;

    int & operator[](std::size_t i) { return data[i]; }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
}
```

Instructor notes:

cleanup (supress all)

rule of 5

demonstrate a[4] = 11

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>
#include <exception>

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) = delete;
    Array(Array && other) = delete;
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) = delete;
    Array & operator=(Array && other) = delete;

    //int & operator[](std::size_t i) { return data[i]; }
    int & operator[](std::size_t i) {
        if (i >= sz)
            throw std::out_of_range("Array::operator[]");
        return data[i];
    }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    try {
        a[4] = 11;
        print_squares(a);
    } catch(const std::out_of_range & ex) {
        std::cerr << "Out of range: " << ex.what() << std::endl;
    }
}
```

Instructor notes:

demonstrate try / catch

discuss “gotcha: throw new” and “gotcha: catch by copy”

discuss and implement int & at() instead

```
Out of range: Array::operator[]
```

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

class Array {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) = delete;
    Array(Array && other) = delete;
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) = delete;
    Array & operator=(Array && other) = delete;

    int & operator[](std::size_t i) { return data[i]; }
    int & at(std::size_t i) {
        if (i >= sz)
            throw std::out_of_range("Array::operator[]");
        return data[i];
    }
    int * begin() const { return &data[0]; }
    int * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Array & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7, 0};
    a[4] = 11;
    print_squares(a);
}
```

Instructor notes:

discuss at() vs operator[]

discuss UB

discuss “trust the programmer”

```
4
9
25
49
121
```

```
#include <iostream>
#include <algorithm>

class Container {
public:
    virtual ~Container() = default;
    virtual int * begin() const = 0;
    virtual int * end() const = 0;
};

class Array : public Container {
public:
    Array(const std::initializer_list<int> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) = delete;
    Array(Array && other) = delete;
    ~Array() override { delete[] data; }
    Array & operator=(const Array & other) = delete;
    Array & operator=(Array && other) = delete;

    int * begin() const override { return &data[0]; }
    int * end() const override { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new int[size]) {}
    size_t sz;
    int * data;
};

void print_squares(const Container & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array a{2, 3, 5, 7};
    print_squares(a);
}
```

Instructor notes:

discuss virtual keyword in subclass

demonstrate override

demonstrate final

demonstrate virtual destructor and = default vs {}

pure interface

… but there is another way to do something similar…

```
4
9
25
49
```

```
#include <iostream>
#include <algorithm>

template <typename T>
class Array {
public:
    Array(const std::initializer_list<T> & list) : Array(list.size()) {
        std::copy(std::cbegin(list), std::cend(list), data);
    }
    Array(const Array & other) = delete;
    Array(Array && other) = delete;
    ~Array() { delete[] data; }
    Array & operator=(const Array & other) = delete;
    Array & operator=(Array && other) = delete;

    T * begin() const { return &data[0]; }
    T * end() const { return &data[sz]; }
private:
    Array(size_t size) : sz(size), data(new T[size]) {}
    size_t sz;
    T * data;
};

template <typename T>
void print_squares(const T & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    Array<int> a{2, 3, 5, 7};
    print_squares(a);
}
```

Instructor notes:

deliberately forget one of the overrides to show that you get an error

discuss STL and standard containers

```
4
9
25
49
```

STL is not object oriented. I think that object orientedness is almost as much of a hoax as Artificial Intelligence. I have yet to see an interesting piece of code that comes from these OO people. In a sense, I am unfair to AI: I learned a lot of stuff from the MIT AI Lab crowd, they have done some really fundamental work: Bill Gosper's Hakmem is one of the best things for a programmer to read. AI might not have had a serious foundation, but it produced Gosper and Stallman (Emacs), Moses (Macsyma) and Sussman (Scheme, together with Guy Steele). I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras - families of interfaces that span multiple types. I find OOP philosophically unsound. It claims that everything is an object. Even if it is true it is not very interesting - saying that everything is an object is saying nothing at all. I find OOP methodologically wrong. It starts with classes. It is as if mathematicians would start with axioms. You do not start with axioms - you start with proofs. Only when you have found a bunch of related proofs, can you come up with axioms. You end with axioms. The same thing is true in programming: you have to start with interesting algorithms. Only when you understand them well, can you come up with an interface that will let them work.

Alexander Stepanov

```cpp
#include <iostream>
#include <array>

template <typename T>
void print_squares(const T & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    std::array<int,4> a{{2, 3, 5, 7}};
    print_squares(a);
}
```

Instructor notes:

notes

```
4
9
25
49
```

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

template <typename T>
void print_squares(const T & a)
{
    for (const auto & v : a)
        std::cout << (v * v) << '\n';
}

int main()
{
    std::vector<int> a{2, 3, 5, 7};
    print_squares(a);
}
```

Instructor notes:

notes

```
4
9
25
49
```

Part 2 - Summary

iterator pattern
countof idiom
user of std::size_t
addressing one element after array
range-for
pointer decay
ptr/len vs iterator vs obejcts (vs ranges)
iterator copy pattern
generic copy function
standard algorithms
constructor and destructor
order of initialization
new with initializer
overloading suscript operator
operator[] vs at()
initializer list
constructor delegation
vexing parse
undefined behavior

shallow vs copy and assignment
default constructors
copy constructor
assignment operator
swap idiom for gutting another object
exception safety
move semantics
supressing/enabling default ctor/operator
std::move
rvalue references
rule of 3 / rule of 5
exception handling
throwing and catching exceptions
"trust the programmer"
class inheritance and virtual functions
override and final specifier
virtual destructor
pure interfaces
STL and standard containers
OOP vs generic programming

Part 3

(algorithms, lamda, concept, concurrency)

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

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

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

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

Instructor notes:

change int to double in for loop, show compilation error

change const_iterator to iterator, show compilation error

```
$ 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 (decltype(log.begin()) it = log.begin(); it != log.end(); ++it)
        transmit_item(*it);
}

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

Instructor notes:

change to auto

```
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 (auto it = log.cbegin(); it != log.cend(); ++it)
        transmit_item(*it);
}

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

Instructor notes:

change to use auto

change to use log.cbegin() and log.cend()

argue that the implementation of transmit_log is not more generic

without doing it, point out that you can now change the item type, but also the container

```
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:

change to use for_each (do not go via range-for)

include <algorithm>

change to std::cbegin() / std::cend()

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)
```

Already seen what happens “under the hood” in Part 2: the Array example

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

Mention that the message is optional in C++17

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

```
37
38
42
45
```

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

struct FooItem {
    FooItem(unsigned short v) : value(v) {}
    explicit operator std::string() const { return std::to_string(value); }
    std::bitset<12> to_duodectet() const { return std::bitset<12>(value); }
    unsigned short value;
};

std::ostream & operator<<(std::ostream & out, const FooItem & i)
{
    return out << i.value << " - " << i.to_duodectet();
}

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

template <typename Log>
void transmit_log(Log && log)
{
    for (const auto & i : log)
        transmit_item(i);
}

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

Instructor notes:

simplify before demonstrating SFINAE

add a struct FooItem

include bitset

add the operator<<

change item_type to FooItem

add the
point out, in this case, useful (but dangerous) implicit ctor

```
24 - 000000011000
20 - 000000010100
37 - 000000100101
42 - 000000101010
34 - 000000100010
45 - 000000101101
38 - 000000100110
```

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

struct FooItem {
    FooItem(unsigned short v) : value(v) {}
    explicit operator std::string() const { return std::to_string(value); }
    std::bitset<12> to_duodectet() const { return std::bitset<12>(value); }
    unsigned short value;
};

std::ostream & operator<<(std::ostream & out, const FooItem & i) {
    return out << i.value << " - " << std::bitset<12>(i.value);
}

template <typename T>
struct has_to_duodectet {
    typedef char yes[2];
    typedef char no[1];
    template <typename I> static yes & test(I * i, decltype(i->to_duodectet()) * = nullptr);
    template <typename> static no & test(...);
    static const bool value = sizeof(test<T>(nullptr)) == sizeof(yes);
};

template <typename LogItem, typename std::enable_if<has_to_duodectet<LogItem>::value>::type * = nullptr>
void transmit_item(LogItem i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log>
void transmit_log(Log && log)
{
    for (const auto & i : log)
        transmit_item(i);
}

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

Instructor notes:

SFINAE

try to explain the has_to_duodectet as you write it

try to explain the use of std::enable_if (perhaps need a simpler example first)

show it works

try to uncomment to_duodectet… show compilation error

also mention:
detected idiom, in c++17
experimental/type_traits

```
24 - 000000011000
20 - 000000010100
37 - 000000100101
42 - 000000101010
34 - 000000100010
45 - 000000101101
38 - 000000100110
```

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

struct FooItem {
    FooItem(unsigned short v) : value(v) {}
    explicit operator std::string() const { return std::to_string(value); }
    std::bitset<12> to_duodectet() const { return std::bitset<12>(value); }
    unsigned short value;
};

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

template <typename Log>
void transmit_log(Log && log)
{
    for (const auto & i : log)
        transmit_item(i);
}

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

Instructor notes:

Remove the SFNIAE and show how to call, in this case, the member function will give kind of similar effect.

```
000000011000
000000010100
000000100101
000000101010
000000100010
000000101101
000000100110
```

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

template <typename T>
concept bool Transmittable() {
    return sizeof(T) == 2;
}

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

template <typename Log>
void transmit_log(Log && log)
{
    for (const auto & i : log)
        transmit_item(i);
}

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

Instructor notes:

remove FooItem

remove <bitset>

clean up transmit_item

```cpp
template <typename T>
concept bool Transmittable() {
    return sizeof(T) == 2;
}

template <typename LogItem> requires Transmittable<LogItem>()

template <Transmittable LogItem>

void transmit_item(Transmittable i)
```

discuss

```cpp
void transmit_log(CollectionOfTransmittables log, UnaryPredicate filter)
```

Conecpts is a gravitational point for C++, everything since C++03 seems to be gradually moving i
this direction.

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

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

template <typename T>
concept bool Transmittable() {
    // check that it is of a transmittable type, but for now...
    return true;
}

template <typename T>
concept bool Iterable() {
    // check that it can be iterated over, but for now...
    return true;
}

template <typename T>
concept bool UnaryFunctionPredicate() {
    // ... check that it takes a value and returns a bool, but for now...
    return true;
}

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

void transmit_log(Iterable log, UnaryFunctionPredicate myfilter)
{
    for (const auto & i : log)
        if (!myfilter(i))
            transmit_item(i);
}

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

Instructor notes:

```
c++ -fconcepts tour.cpp && ./a.out
37
42
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:

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 pattern

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
```

Part 3 - Summary

iterators
decltype and auto
STL algorithms
imperative programming style
declarative programming style
for_each, sort, remove_if, erase
"unexpected behavior"
call by value
call by reference
passing ownership
rvalue reference
move semantics
temporary objects
working with "ranges"
erase-remove idiom
programming is communication
injecting strategies
parameterize from above
filters and predicates
function objects (aka functors)
unamed functions
lambda expressions
closures
capture by value
capture by reference
generalized lambda capture

passing callable objects
dependent names
type independent code
generic programming
type traits
static asserts
explicit (full) template specialization
partial template specialization
SFINAE
std::enable_if
detected idiom
experimental type_traits
concepts
chrono
threads
promise
future
higher order parallelism
async
mutex
chrono literals
acquire / release pattern
RAII - resource acquisition is initialization
lock quards
atomic
C and C++

A Tour of Modern C++

In this fast-paced course we will start from scratch and relearn C++ with modern syntax and semantics. Among other things you will learn (at least something) 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, C++14 and new stuff coming with C++17 and later

This course is aimed at experienced programmers that would like to learn how to write, or at least understand, modern C++. Ideally you should have some experience with either C, old-school C++, Python and/or Java.

!

Todo:

create a static version of Array, eg Array<int, 4>
Find out about steady_clock issue
Lambdas and globals
warning about rule of three / rule of five
how to write classes that are destroyable/copyable/movable
something about functional programming in C++
more about constexpr