C++ for C Programmers

Victor Eijkhout

2025 June 02–06

THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center

Introduction

1. Stop Coding C!

1. C++ is a more structured and safer variant of C:
   There are very few reasons not to switch to C++.
2. C++ (almost) contains C as a subset.
   So you can use any old mechanism you know from C
   However: where new and better mechanisms exist, stop using the old style C-style idioms.
   `https://www.youtube.com/watch?v=YnWhqhNdYyk`

2. In this course

1. Object-oriented programming.
2. New mechanisms that replace old ones:
   I/O, strings, arrays, pointers, random, union.
3. Other new mechanisms:
   exceptions, namespaces, closures, templating

I'm assuming that you know how to code C loops and functions and you understand what structures and pointers are.

3. About this course

Slides and codes are from my open source text book:

`https://theartofhpc.com/isp.html`

4. General note about syntax

Many of the examples in this lecture use the C++17/20/23 (sometimes C++26) standard.

```
icpc    -std=c++23 yourprogram.cxx
g++     -std=c++23 yourprogram.cxx
clang++ -std=c++23 yourprogram.cxx
```

There is no reason not to use that all the time:

```
alias icpc='icpc -std=c++23'
et cetera
```

(Each compiler has a different default version)

5. Build with Cmake

```
cmake_minimum_required( VERSION 3.20 )
project( ${PROGRAM_NAME} VERSION 1.0 )

add_executable( ${PROGRAM_NAME} ${PROGRAM_NAME}.cpp )
target_compile_features( ${PROGRAM_NAME}
    PRIVATE cxx_std_23 )
```

6. C++ standard

- C++98/C++03: ancient.
  There was a lot wrong or not-great with this.
- C++11/14/17: ‘modern’ C++.
  What everyone uses.
- C++20/23: ‘post-modern’ C++.
  Ratified, but only partly implemented.
- C++26: being defined.

Minor enhancements

7. Just to have this out of the way

- There is a `bool` type with values `true`, `false`
- Single line comments:

```
int x = 1; // set to one
```

- More readable than typedef:

```
using Real = float;
Real f( Real x ) { /* ... */ };
Real g( Real x,Real y ) { /* ... */ };
```

Change your mind about `float`/`double` in one stroke.
Abbrev for complicated types.

8. Initializer statement

Loop variable can be local (also in C99):

```
for (int i=0; i<N; i++) // do whatever
```

Similar in conditionals and switch:

```
// basic/ifinit.cpp
if ( char c = getchar(); c!='a' )
  cout << "Not an a, but: "
       << c << '\n';
else
  cout << "That was an a!"
       << '\n';
```

(strangely not in `while`)

9. Simple I/O

Headers:

```
#include <iostream>
using std::cin;
using std::cout;
```

Ouput:

```
int main() {
  int plan=4;
  cout << "Plan " << plan << " from outer space" << "\n";
```

Input:

```
int i;
cin >> i;
```

(string input limited to no-spaces)

10. Hello world!

Let's do a 'hello world', using `std::cout`:

Ok for now:

```cpp
#include <iostream>
using namespace std;
int main() {
  cout << "Hello world\n";
}
```

Better:

```cpp
#include <iostream>
using std::cout;
int main() {
  cout << "Hello world\n";
}
```

11. Hello world again!

Let's do a 'hello world', using `std::format`:

```
#include <iostream>
#include <format>
int main() {
  std::cout << std::format("Hello world\n");
}
```

In C++23:

```
#include <print>
std::println("Hello world");
```

Functions

12. Big and small changes

- Minor changes: default values on parameters, and polymorphism.
- Big change: use references instead of addresses for argument passing.

Parameter passing

13. C++ references different from C

- C does not have an actual pass-by-reference:
  C mechanism passes address by value.
- C++ has ‘references’, which are different from C addresses.
- The `&` ampersand is used, but differently.
- Asterisks are out:
  rule of thumb for now,
  if you find yourself writing asterisks, you’re not writing C++.
  (however, there are exceptions and advanced uses)

14. Reference

A reference is indicated with an ampersand in its definition, and it acts as an alias of the thing it references.

Code:

```
// basic/ref.cpp
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << '\n';
i *= 2;
cout << i << "," << ri << '\n';
ri -= 3;
cout << i << "," << ri << '\n';
```

Output:

```
5,5
10,10
7,7
```

(You will not use references often this way.)

15. Create reference by initialize

Correct:

```
float x{1.5};
float &xref = x;
```

Not correct:

```
float x{1.5};
float &xref; // WRONG: needs to initialized immediately
xref = x;

float &threeref = 3; // WRONG: only reference to `lvalue'
```

16. Reference vs pointer

- There are no ‘null’ references.
  (`nullptr` has nothing to do with references.)
- References are bound when they are created.
- You can not change what a reference is bound to;
  a pointer target can change.

17. Parameter passing by reference

The function parameter `n` becomes a reference to the variable `i` in the main program:

```
void f(int &n) {
  n = /* some expression */ ;
};
int main() {
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

18. Pass by reference example 1

Code:

```
// basic/setbyref.cpp
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << '\n';
```

Output:

```
5
```

Compare the difference with leaving out the reference.

19. Const ref parameters

```
void f( const int &i ) { .... }
```

- Pass by reference: no copying, so cheap
- Const: no accidental altering.
- Especially useful for large objects.

Exercise 1

Write a `void` function *swap* of two parameters that exchanges the input values:

Code:

```
// func/swap.cpp
int i=1,j=2;
cout << i << "," << j << '\n';
swap(i,j);
cout << i << "," << j << '\n';
```

Output:

```
1,2
2,1
```

Optional exercise 2

Write a divisibility function that takes a number and a divisor, and gives:

- a `bool` return result indicating that the number is divisible, and
- a remainder as output parameter.

Code:

```
// func/divisible.cpp
cout << number;
if (is_divisible(number,
          divisor,remainder))
  cout << " is divisible by ";
else
  cout << " has remainder "
     << remainder << " from ";
cout << divisor << '\n';
```

Output:

```
8 has remainder 2
    ↪from 3
8 is divisible by 4
```

More about functions

20. Default arguments

Functions can have default argument(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

21. Useful idiom

Don't trace a function unless I say so:

```
void dosomething(double x,bool trace=false) {
  if (trace) // report on stuff
};
int main() {
  dosomething(1); // this one I trust
  dosomething(2); // this one I trust
  dosomething(3,true); // this one I want to trace!
  dosomething(4); // this one I trust
  dosomething(5); // this one I trust
```

22. Polymorphic functions

You can have multiple functions with the same name:

```
double average(double a,double b) {
  return (a+b)/2; }
double average(double a,double b,double c) {
  return (a+b+c)/3; }
```

Distinguished by type or number of input arguments: can not differ only in return type.

```
int f(int x);
string f(int x); // DOES NOT WORK
```

Object-Oriented Programming

Classes

23. Definition of object/class

An object is an entity that you can request to do certain things. These actions are the *methods*, and to make these possible the object probably stores data, the *members*.

When designing a class, first ask yourself:
'what functionality should the objects support'.

A class is a user-defined type; an object is an instance of that type.

24. Running example

We are going to build classes for points/lines/shapes in the plane.

```
class Point {
    /* stuff */
};
int main () {
  Point p; /* stuff */
}
```

Exercise 3

Thought exercise: what are some of the actions that a point object should be capable of?

25. Object functionality

Small illustration: point objects.

Code:

```
// object/functionality.cpp
Point p(1.,2.);
cout << "distance to origin "
     << p.distance_to_origin()
     << '\n';
p.scaleby(2.);
cout << "distance to origin "
     << p.distance_to_origin()
     << '\n'
     << "and angle " << p.angle()
     << '\n';
```

Output:

```
distance to origin
     ↪2.23607
distance to origin
     ↪4.47214
and angle 1.10715
```

Note the 'dot' notation.

Exercise 4

Thought exercise:
What data does the object need to store to be able to calculate angle and distance to the origin?
Is there more than one possibility?

26. The object workflow

- First define the class, with data and function members:

```
class MyObject {
  // define class members
  // define class methods
};
```

  (details later) typically before the `main`.

- You create specific objects with a declaration

```
MyObject
  object1( /* .. */ ),
  object2( /* .. */ );
```

- You let the objects do things:

```
object1.do_this();
x = object2.do_that( /* ... */ );
```

27. Construct an object

The declaration of an object `x` of class `Point`; the coordinates of the point are initially set to `1.5,2.5`.

```
Point x(1.5, 2.5);
```

```
class Point {
private: // data members
  double x,y;
public: // function members
  Point
    (double x_in,double y_in){
      x = x_in; y = y_in;
  };
  /* ... */
};
```

Use the constructor to create an object of a class: function with same name as the class.
(but no return type!)

28. Private and public

Best practice we will use:

```
class MyClass {
private:
  // data members
public:
  // methods
}
```

- Data is private: not visible outside of the objects.
- Methods are public: can be used in the code that uses objects.
- You can have multiple private/public sections, in any order.

Methods

29. Class methods

Definition and use of the `distance` function:

Code:

```
// geom/pointclass.cpp
class Point {
private:
  float x,y;
public:
  Point(float in_x,float in_y) {
    x = in_x; y = in_y; };
  float distance_to_origin() {
    return sqrt( x*x + y*y );
  };
};
    /* ... */
  Point p1(1.0,1.0);
  float d = p1.distance_to_origin();
  cout << "Distance to origin: "
       << d << '\n';
```

Output:

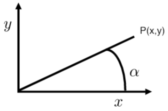
```
Distance to origin:
    ↪1.41421
```

30. Class methods

- Methods look like ordinary functions,
- except that they can use the data members of the class, for instance `x`,`y`;
- Methods can only be used on an object with the 'dot' notation. They are not independently defined.

Exercise 5

Add a method *`angle`* to the *`Point`* class. How many parameters does it need?



Hint: use the function *`atan`* or *`atan2`*.

You can base this off the file `pointclass.cpp` in the repository

Exercise 6

Make a class `GridPoint` for points that have only integer coordinates. Implement a function `manhattan_distance` which gives the distance to the origin counting how many steps horizontal plus vertical it takes to reach that point.

31. Food for thought: constructor vs data

The arguments of the constructor imply nothing about what data members are stored!

Example: create a point in *x*,*y* Cartesian coordinates, but store *r*,*theta* polar coordinates:

```
#include <cmath>
class Point {
private: // members
  double r,theta;
public: // methods
  Point( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan2(y,x);
  }
```

Note: no change to outward API.

Exercise 7

Discuss the pros and cons of this design:

```
class Point {
private:
  double x,y,r,theta;
public:
  Point(double xx,double yy) {
    x = xx; y = yy;
    r = // sqrt something
    theta = // something trig
  };
  double angle() { return theta; };
};
```

32. Data access in methods

You can access data members of other objects of the same type:

```
class Point {
private:
  double x,y;
public:
 void flip() {
    Point flipped;
    flipped.x = y; flipped.y = x;
    // more
  };
};
```

(Normally, data members should not be accessed directly from outside an object)

Exercise 8

Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

33. Methods that alter the object

For instance, you may want to scale a vector by some amount:

Code:

```
// geom/pointscaleby.cpp
class Point {
    /* ... */
  void scaleby( float a ) {
    x *= a; y *= a; };
    /* ... */
};
    /* ... */
  Point p1(1.,2.);
  cout << "p1 to origin "
       << p1.distance_to_origin()
       << '\n';
  p1.scaleby(2.);
  cout << "p1 to origin "
       << p1.distance_to_origin()
       << '\n';
```

Output:

```
p1 to origin 2.23607
p1 to origin 4.47214
```

Data initialization

34. Member default values

Class members can have default values, just like ordinary variables:

```
class Point {
private:
  float x=3., y=.14;
public:
  // et cetera
}
```

Each object will have its members initialized to these values.

35. Data initialization

The naive way:

```
class Point {
private:
  double x,y;
public:
  Point( double in_x,
         double in_y ) {
    x = in_x; y = in_y;
  };
```

The preferred way:

```
// geom/pointinit.cpp
class Point {
private:
  float x,y;
public:
  Point( float in_x,
         float in_y )
    : x(in_x),y(in_y) {
  }
```

Explanation later. It's technical.

Interaction between objects

36. Methods that create a new object

Code:

```
// geom/pointscale.cpp
class Point {
    /* ... */
  Point scale( float a ) {
    Point scaledpoint( x*a, y*a );
    return scaledpoint;
  };
    /* ... */
  println("p1 to origin {:.5}",
          p1.dist_to_origin());
  Point p2 = p1.scale(2.);
  println("p2 to origin {:.5}",
          p2.dist_to_origin());
```

Output:

```
p1 to origin 2.2361
p2 to origin 4.4721
```

Note the 'anonymous `Point` object' in the `scale` method.

37. Anonymous objects

Create a point by scaling another point:

```
new_point = old_point.scale(2.81);
```

Two ways of handling the `return` statement of the `scale` method:

Naive:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

Creates point, copies it to `new_point`

Better:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
  return Point( x*a, y*a );
};
```

Creates point, moves it directly to `new_point`

'move semantics' and 'copy elision':
compiler is pretty good at avoiding copies

Exercise 9

Write a method `translated` that, given a `Point` and two `floats`, returns the point translated by those amounts:

Code:

```
// geom/halfway.cpp
Point p1( 1.5,2.5 );
cout << p1.stringified() << '\n';
float x=.2, y=.3;
auto p2 = p1.translated(x,y);
cout << "by: " << x << "," << y <<
     '\n';
cout << p2.stringified() << '\n';
```

Output:

```
(1.5,2.5)
by: 0.2,0.3
(1.7,2.8)
```

Optional exercise 10

Write a method `halfway` that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p+q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these.
(Later you will learn about operator overloading.)

How would you print out a `Point` to make sure you compute the halfway point correctly?

38. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
// object/stream.cpp
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

int main() {
  Stream ints;
  cout << "Next: "
       << ints.next() << '\n';
  cout << "Next: "
       << ints.next() << '\n';
  cout << "Next: "
       << ints.next() << '\n';
```

Output:

```
Next: 0
Next: 1
Next: 2
```

39. Preliminary to the following exercise

A prime number generator has:
an API of just one function: `nextprime`

To support this it needs to store:
an integer `last_prime_found`

Programming Project Exercise 11

Write a class *primegenerator* that contains:

- Methods *number_of_primes_found* and *nextprime*;
- Also write a function *isprime* that does not need to be in the class.

Your main program should look as follows:

```
// primes/6primesbyclass.cpp
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

Programming Project Exercise 12

Write a program to test the Goldbach conjecture for the even numbers up to a bound that you read in.

First formulate the quantor structure of this statement, then translate that top-down to code, using the generator you developed above.

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e,p,q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

40. A Goldbach corollary

The Goldbach conjecture says that every even number $2n$ (starting at 4), is the sum of two primes $p+q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r\,\mathrm{prime}}\exists_{p,q\,\mathrm{prime}}\colon r = (p+q)/2 \text{ is prime.}$$

We now have the statement that each prime number is the average of two other prime numbers.

Programming Project Exercise 13

Write a program that tests this. You need at least one loop that tests all primes r; for each r you then need to find the primes p, q that are equidistant to it.

Use your prime generator. Do you use two generators for this, or is one enough? Do you need three, for p, q, r?

For each r value, when the program finds the p, q values, print the p, q, r triple and move on to the next r.

Class inheritance: is-a

41. Examples for base and derived cases

General *FunctionInterpolator* class with method *value_at*. Derived classes:

- *LagranceInterpolator* with *add_point_and_value*;
- *HermiteInterpolator* with *add_point_and_derivative*;
- *SplineInterpolator* with *set_degree*.

42. General case, special case

You can have classes where an object of one class is a special case of the other class. You declare that as

```
class General {
protected: // note!
 int g;
public:
 void general_method() {};
};

class Special : public General {
public:
  void special_method() { g = ... };
};
```

43. Inheritance: derived classes

Derived class `Special` *inherits* methods and data from base class `General`:

```
int main() {
  Special special_object;
  special_object.general_method();
  special_object.special_method();
}
```

Members of the base class need to be `protected`, not `private`, to be inheritable.

44. Constructors

When you run the special case constructor, usually the general constructor needs to run too. Here we invoke it explicitly:

```
class General {
public:
  General( double x,double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
};
```

45. Access levels

Methods and data can be

- `private`, because they are only used internally;
- `public`, because they should be usable from outside a class object, for instance in the main program;
- `protected`, because they should be usable in derived classes.

Exercise 14

Take your code where a `Rectangle` was defined from one point, width, and height.

Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

First ask yourself: what should the constructor of a `Square` look like?

Operator overloading

46. Better syntax

Operations that 'feel like arithmetic'"

So far:

```
Point p3 = p1.add(p2);
Point p4 = p3.scale(2.5);
```

Improved:

```
Point p3 = p1+p2;
Point p4 = p3*2.5;
```

This is possible because you can *overload* the *operators*. For instance,

```
// geom/overload.cpp
Point operator*( float f ) {
  return Point( x*f,y*f );
}
```

47. Operator overloading

Syntax:

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

Code:

```
// geom/pointscale.cpp
Point Point::operator*(float f) {
    return Point(f*x,f*y);
};
    /* ... */
  println("p1 to origin {:.5}",
          p1.dist_to_origin());
  Point scale2r = p1*2.;
  println("scaled right: {}",
          scale2r.dist_to_origin());
  // ILLEGAL Point scale2l = 2.*p1;
```

Output:

```
p1 to origin 2.2361
scaled right:
     ↪4.472136
```

Exercise 15

Define the plus operator between *Point* objects. The declaration is:

```
Point operator+(Point q);
```

You can base this off the file `overload.cpp` in the repository

Exercise 16

Rewrite the `halfway` method of exercise 10 and replace the `add` and `scale` functions by overloaded operators.

Hint: for the `add` function you may need '`this`'.

48. Functor example

Simple example of overloading parentheses:

Code:

```
// object/functor.cpp
class IntPrintFunctor {
public:
  void operator()(int x) {
    println("{}",x);
  }
};
    /* ... */
  IntPrintFunctor intprint;
  intprint(5);
```

Output:

```
5
```

Exercise 17

Evaluate a linear function:

Using method:

```
// geom/overload.cpp
LinearFunction line(p1,p2);
cout << "Value at 4.0: "
     << line.evaluate_at(4.0)
     << '\n';
```

using operator:

```
// geom/overload.cpp
float y = line(4.0);
cout << y << '\n';
```

Write the appropriate overloaded operator.

You can base this off the file `overload.cpp` *in the repository*

Vectors

49. C++ Vectors are better than C arrays

Vectors are fancy arrays. They are easier and safer to use:

- They know what their size is.
- Bound checking.
- Freed when going out of scope: no memory leaks.
- Dynamically resizable.

In C++ you never have to `malloc` again.
(Rarely ever `new`.)

50. Vectors, the new and improved arrays

- C array/pointer equivalence is silly
- C++ vectors are just as efficient
- ... and way easier to use.

Don't use use explicitly allocated arrays anymore

```
double *array = (double*) malloc(n*sizeof(double)); // No!
double *array = new double[n]; // please don't (rare exceptions)
```

51. Vector definition

Definition and/or initialization:

```
#include <vector>
using std::vector;

vector<type> name;
vector<type> name(size);
vector<type> name(size,init_value);
```

where

- `vector` is a keyword,
- *`type`* (in angle brackets) is any elementary type or class name,
- *`name`* of the vector is up to you, and
- `size` is the (initial size of the vector). This is an integer, or more precisely, a `size_t` parameter.
- Initialize all elements to *`init_value`*.
- If no default given, zero is used for numeric types.

52. Accessing vector elements

Square bracket notation (zero-based):

Code:

```
// array/assign.cpp
vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

Output:

```
4,8
```

With bound checking:

Code:

```
// array/assign.cpp
vector<int> numbers = {1,4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << '\n';
```

Output:

```
4,8
```

53. Vector elements out of bounds

Square bracket notation:

Code:

```
// array/assignoutofbound.cpp
vector<int> foo(25);
vector<int> numbers = {1,4};
numbers[-1] += 3;
numbers[2] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

Output:

```
1,4
```

With bound checking:

Code:

```
// array/assignoutofbound.cpp
vector<int> numbers = {1,4};
numbers.at(-1) += 3;
numbers.at(2) = 8;
cout << numbers.at(0) << ","
```

Output:

```
libc++abi:
    ↪terminating
    ↪with uncaught
    ↪exception of
    ↪type
```

54. Short vectors

Short vectors can be created by enumerating their elements:

```
// array/shortvector.cpp
#include <vector>
using std::vector;

int main() {
  vector<int> evens{0,2,4,6,8};
  vector<float> halves = {0.5, 1.5, 2.5};
  auto halfloats = {0.5f, 1.5f, 2.5f};
  cout << evens.at(0)
       << " from " << evens.size()
       << '\n';
  return 0;
}
```

Exercise 18

1. Take the above snippet, compile, run.
2. Add a statement that alters the value of a vector element. Check that it does what you think it does.
3. Add a vector of the same length as the `evens` vector, containing odd numbers which are the even values plus 1?

You can base this off the file `shortvector.cpp` in the repository

55. Range over elements

A range-based for loop gives you directly the element values:

```
vector<float> my_data(N);
/* set the elements somehow */;
for ( float e : my_data )
  // statement about element e
```

Here there are no indices because you don't need them.

56. Range over elements, version 2

Same with auto instead of an explicit type for the elements:

```
for ( auto e : my_data )
  // same, with type deduced by compiler
```

57. Range over elements

Finding the maximum element

Code:

```
// array/dynamicmax.cpp
vector<int> numbers = {1,4,2,6,5};
int tmp_max = -2000000000;
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max
     << " (should be 6)" << '\n';
```

Output:

```
Max: 6 (should be 6)
```

Exercise 19

Find the element with maximum absolute value in a vector. Use:

```
vector<int> numbers = {1,-4,2,-6,5};
```

Hint:

```
#include <cmath>
..
absx = abs(x);
```

Exercise 20

Indicate for each of the following vector operations whether you prefer to use an indexed loop or a range-based loop. Give a short motivation.

- Count how many elements of a vector are zero.
- Find the location of the last zero.

58. Range over vector denotation

Code:

```
// array/rangedenote.cpp
for ( auto i : {2,3,5,7,9} )
  cout << i << ",";
cout << '\n';
```

Output:

```
2,3,5,7,9,
```

59. Range over elements by reference

Range-based loop indexing makes a copy of the vector element. If you want to alter the vector, use a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
// array/vectorrangeref.cpp
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2)
     << '\n';
```

Output:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

60. Example: multiplying elements

Example: multiply all elements by two:

Code:

```
// array/vectorrangeref.cpp
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2)
     << '\n';
```

Output:

```
6.6
```

61. Indexing the elements

You can write an indexed for loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
  // statement about index i
```

Example: find the maximum element in the vector, and where it occurs.

Code:

```
// array/vectoridxmax.cpp
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i=0; i<numbers.size(); ++i) {
  int v = numbers.at(i);
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << '\n';
```

Output:

```
Max: 6.6 at
    ↪index: 3
```

62. On finding the max and such

For many such short loops there are ‘algorithms’; see later.

63. A philosophical point

Conceptually, a `vector` can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

Exercise 21

Find the location of the first negative element in a vector.

Which mechanism do you use?

Exercise 22

Create a `vector` x of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in L_2 norm and check the correctness of your calculation, that is,

1. Compute the L_2 norm of the vector:
$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$
2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

64. Vector copy

Vectors can be copied just like other datatypes:

Code:

```
// array/vectorcopy.cpp
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << '\n';
```

Output:

```
3.5,7
```

Note: contents copied, not just pointer.

65. Vector methods

A vector is an object, with methods.

Given `vector<sometype> x`:

- Get elements, including bound checking, with `ar.at(3)`.
  Note: zero-based indexing.
  (also get elements with `ar[3]`: see later discussion.)
- Size: `ar.size()`.
- Other functions: `front`, `back`, `empty`, `push_back`.
- With iterators (see later): `insert`, `erase`

66. Your first encounter with templates

`vector` is a 'templated class': `vector<X>` is a vector-of-X.

Code behaves as if there is a class definition for each type:

```
class vector<int> {
public:
  size(); at(); // stuff
}
```

```
class vector<float> {
public:
  size(); at(); // stuff
}
```

Actual mechanism uses templating: the type is a parameter to the class definition.

Dynamic behaviour

67. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

```
// array/vectorend.cpp
vector<int> mydata(5,2);
// last element:
cout << mydata.back()
     << '\n';
mydata.push_back(35);
cout << mydata.size()
     << '\n';
// last element:
cout << mydata.back()
     << '\n';
```

Output:

```
2
6
35
```

Similar functions: `pop_back`, `insert`, `erase`.

Flexibility comes with a price.

68. When to push back and when not

Known vector size:

```
int n = get_inputsize();
vector<float> data(n);
for ( int i=0; i<n; i++ ) {
  auto x = get_item(i);
  data.at(i) = x;
}
```

Unknown vector size:

```
vector<float> data;
float x;
while ( next_item(x) ) {
  data.push_back(x);
}
```

If you have a guess as to size: `data.reserve(n)`.

(Issue with array-of-object: in left code, constructors are called twice.)

69. Filling in vector elements

You can push elements into a vector:

```
// array/arraytime.cpp
vector<int> flex;
/* ... */
for (int i=0; i<LENGTH; ++i)
  flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat.at(i) = i;
```

70. Filling in vector elements

With subscript:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat[i] = i;
```

You can also use `new` to allocate*:

```
// array/arraytime.cpp
int *stat = new int[LENGTH];
/* ... */
for (int i=0; i<LENGTH; ++i)
  stat[i] = i;
```

*Considered bad practice. Do not use.

71. Timing the ways of filling a vector

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

72. Size and capacity

Grow by pushing elements:

Code:

```
// array/grow.cpp
  vector<int> data;
  const int up=10;
  for ( int i=0; i<=up; ++i ) {
    cout << "size=" << data.size()
         << ", capacity=" << data.
     capacity()
         << '\n';
    if (i==up) break;
    data.push_back(i);
  }
```

Output:

```
size=0, capacity=0
size=1, capacity=1
size=2, capacity=2
size=3, capacity=4
size=4, capacity=4
size=5, capacity=8
size=6, capacity=8
size=7, capacity=8
size=8, capacity=8
size=9, capacity=16
size=10, capacity=16
```

73. Size and capacity

Grow by resizing:

Code:

```
// array/grow.cpp
cout << "Resizing:\n";
data.resize( data.size()+7 );
cout << "size=" << data.size()
     << ", capacity=" << data.capacity()
     << '\n';
```

Output:

```
Resizing:
size=17, capacity=20
```

Vectors and functions

74. Vector as function return

You can have a vector as return type of a function.
Example: this function creates a vector, with the first element set to the size:

Code:

```
// array/vectorreturn.cpp
vector<int> make_vector(int n) {
  vector<int> x(n);
  x.at(0) = n;
  return x;
}
    /* ... */
  vector<int> x1 = make_vector(10);
  // "auto" also possible!
  cout << "x1 size: "
       << x1.size() << '\n';
  cout << "zero element check: "
       << x1.at(0) << '\n';
```

Output:

```
x1 size: 10
zero element check:
     ↪10
```

75. Vector as function argument

You can pass a vector to a function:

```
double slope( vector<double> v ) {
  return v.at(1)/v.at(0);
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

76. Vector pass by value example

Code:

```
// array/vectorpassnot.cpp
void set0
  ( vector<float> v,float x )
{
  v.at(0) = x;
}
    /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

Output:

```
3.5
```

- Vector is copied
- 'Original' in the calling environment not affected
- Cost of copying?

77. Vector pass by reference

If you want to alter the vector, you have to pass by reference:

Code:

```
// array/vectorpassref.cpp
void set0
  ( vector<float> &v,float x )
{
  v.at(0) = x;
}
    /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << '\n';
```

Output:

```
4.6
```

- Parameter vector becomes alias to vector in calling environment ⇒ argument *can* be affected.
- No copying cost
- What if you want to avoid copying cost, but need not alter the argument?

78. Vector pass by const reference

Passing a vector that does not need to be altered:

```
int f( const vector<int> &ivec ) { ... }
```

- Zero copying cost
- Not alterable, so: safe!
- (No need for pointers!)

Exercise 23

Revisit exercise 22 and introduce a function for computing the L_2 norm.

Strings

79. String declaration

```cpp
#include <string>
using std::string;

// .. and now you can use `string'
```

(Do not use the C legacy mechanisms.)

80. String creation

A string variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

81. Quotes in strings

You can escape a quote, or indicate that the whole string is to be taken literally:

Code:

```
// string/quote.cpp
string
  one("a b c"),
  two("a \"b\" c"),
  three( R"("a ""b """c)" );
cout << one << '\n';
cout << two << '\n';
cout << three << '\n';
```

Output:

```
a b c
a "b" c
"a ""b """c
```

82. Concatenation

Strings can be *concatenated*:

Code:

```
// string/strings.cpp
string my_string, space{" "};
my_string = "foo";
my_string += space + "bar";
cout << my_string << ": " <<
     my_string.size() << '\n';
```

Output:

```
foo bar: 7
```

83. String indexing

You can query the *size*:

Code:

```
// string/strings.cpp
string five_text{"fiver"};
cout << five_text.size() << '\n';
```

Output:

```
5
```

or use subscripts:

Code:

```
// string/stringsub.cpp
string digits{"0123456789"};
cout << "char three: "
     << digits[2] << '\n';
cout << "char four : "
     << digits.at(3) << '\n';
```

Output:

```
char three: 2
char four : 3
```

84. Ranging over a string

Same as ranging over vectors.

Range-based for:

Code:

```
// string/stringrange.cpp
cout << "By character: ";
for ( char c : abc )
  cout << c << " ";
cout << '\n';
```

Output:

```
By character: a b c
```

Ranging by index:

Code:

```
// string/stringrange.cpp
string abc = "abc";
cout << "By character: ";
for (int ic=0; ic<abc.size(); ic++)
  cout << abc[ic] << " ";
```

Output:

```
By character: a b c
```

85. Range with reference

Range-based for makes a copy of the element
You can also get a reference:

Code:

```
// string/stringrange.cpp
for ( char &c : abc )
  c += 1;
cout << "Shifted: " << abc << '\n';
```

Output:

```
Shifted: bcd
```

Quiz 1

True or false?

1. `'0'` is a valid value for a `char` variable single-quote 0 is a valid char+
2. `"0"` is a valid value for a `char` variable double-quote 0 is a valid char+
3. `"0"` is a valid value for a `string` variable double-quote 0 is a valid string+
4. `'a'+'b'` is a valid value for a `char` variable adding single-quote chars is a valid char+

Exercise 24

The oldest method of writing secret messages is the Caesar cipher. You would take an integer s and rotate every character of the text over that many positions:

$$s \equiv 3\colon \text{"acdz"} \Rightarrow \text{"dfgc".}$$

Write a program that accepts an integer and a string, and display the original string rotated over that many positions.

86. More vector methods

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

Code:

```
// string/strings.cpp
string five_chars;
cout << five_chars.size() << '\n';
for (int i=0; i<5; ++i)
  five_chars.push_back(' ');
cout << five_chars.size() << '\n';
```

Output:

```
0
5
```

Methods only for `string`: `find` and such.

`http://en.cppreference.com/w/cpp/string/basic_string`

Exercise 25

Write a function to print out the digits of a number: `156` should print `one five six`. You need to convert a digit to a string first; can you think of more than one way to do that?

Start by writing a program that reads a single digit and prints its name.

For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

Optional exercise 26

Write a function to convert an integer to a string: the input `215` should give `two hundred fifteen`, et cetera.

Binary

87. Binary I/O

Binary output: write your data byte-by-byte from memory to file.
(Why is that better than a printable representation?)

Code:

```
// io/fiobin.cpp
cout << "Writing: " << x << '\n';
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
file_out.write
  (reinterpret_cast<char*>(&x),
   sizeof(double));
file_out.close();
```

Output:

```
Writing:
     ↪0.841471
```

`write` takes an address and the number of bytes.

88. Binary I/O

Binary output: write your data byte-by-byte from memory to file. (Why is that better than a printable representation?)

Code:

```
// io/fiobin.cpp
cout << "Writing: " << x << '\n';
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
file_out.write
  (reinterpret_cast<char*>(&x),
   sizeof(double));
file_out.close();
```

Output:

```
Writing:
      ↪0.841471
```

`write` takes an address and the number of bytes.

Output an object

89. String stream

Like cout (including conversion from quantity to string), but to object, not to screen.

- Use the `<<` operator to build it up; then
- use the *str* method to extract the string.

```
#include <sstream>
stringstream s;
s << "text" << 1.5;
cout << s.str() << endl;
```

90. String an object, 1

Define a function that yields a string representing the object:

```
#include <sstream>
using std::stringstream;
```

```
// geom/pointfunc.cpp
string as_string() {
  stringstream ss;
  ss << "(" << x << "," << y << ")"
     ;
  return ss.str();
};
```

```
// geom/pointfunc.cpp
string as_fmt_string() {
  auto ss = format("({},{})",x,y);
  return ss;
};
```

91. String an object, 2

Redefine the less-less operator:

```
// geom/pointfunc.cpp
std::ostream& operator<<
(std::ostream &out,Point &p) {
  out << p.as_string(); return out;
};
```

Use:

```
// geom/pointfunc.cpp
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << "\n";
```

I/O

92. Default unformatted output

Code:

```
// io/io.cpp
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << '\n';
```

Output:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

93. Fancy formatting

1. Header: `iomanip`: manipulation of `cout`
2. std::format: looks like `printf` and Python's `print(f"stuff")`.

94. Format arguments

Argument mechanism:

- Arguments indicated by curly braces in the format string;
- braces can contain numbers (and modifiers, see next)

Code:

```
// iofmt/fmtbasic.cpp
println("{}",2);
string hello_string = format
  ("{} {}!","Hello","world");
cout << hello_string << '\n';
println
  ("{0}, {0} {1}!",
   "Hello","world");
```

Output:

```
2
Hello world!
Hello, Hello world!
```

95. Reserve space

You can specify the number of positions, and the output is right aligned in that space by default:

Code:

```
// io/width.cpp
#include <iomanip>
using std::setw;
    /* ... */
  cout << "Width is 6:" << '\n';
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setw(6) << i << '\n';
  cout << '\n';

  // 'setw' applies only once:
  cout << "Width is 6:" << '\n';
  cout << ">"
       << setw(6) << 1 << 2 << 3 <<
    '\n';
  cout << '\n';
```

Output:

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
>     123
```

96. Right align

Right-align with > character and width:

Code:

```
// iofmt/fmtlib.cpp
for (int i=10; i<200000000; i*=10)
  fmt::print("{:>6}\n",i);
//cout << format("{:>6}\n",i);
```

Output:

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
```

97. Padding character

Normally, padding is done with spaces, but you can specify other characters:

Code:

```
// io/formatpad.cpp
#include <iomanip>
using std::setfill;
using std::setw;
    /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.')
         << setw(6) << i
         << '\n';
```

Output:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

98. Left alignment

Instead of right alignment you can do left:

Code:

```
// io/formatleft.cpp
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
    /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.')
         << setw(6) << i << '\n';
```

Output:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

99. Padding character

Other than space for padding:

Code:

```
// iofmt/fmtlib.cpp
for (int i=10; i<200000000; i*=10)
  fmt::print("{0:.>6}\n",i);
//cout << format("{0:.>6}\n",i);
```

Output:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
```

100. Number base

Finally, you can print in different number bases than 10:

Code:

```
// io/format16.cpp
#include <iomanip>
using std::setbase;
using std::setfill;
    /* ... */
  cout << setbase(16)
       << setfill(' ');
  for (int i=0; i<16; ++i) {
    for (int j=0; j<16; ++j)
      cout << i*16+j << " " ;
    cout << '\n';
  }
```

Output:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a
```

101. Number bases

Code:

```
// iofmt/fmtlib.cpp
fmt::print
  ("{0} = {0:b} bin\n",17);
fmt::print
  ("    = {0:o} oct\n",17);
fmt::print
  ("    = {0:x} hex\n",17);
```

Output:

```
17 = 10001 bin
   = 21 oct
   = 11 hex
```

Exercise 27

Make the first line in the above output align better with the other lines:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

102. Fixed point precision

Fixed precision applies to fractional part:

Code:

```
// io/fix.cpp
x = 1.234567;
cout << fixed;
for (int i=0; i<10; ++i) {
  cout << setprecision(4) << x << '\
    n';
  x *= 10;
}
```

Output:

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

(Notice the rounding)

Exercise 28

Use integer output to print real numbers aligned on the decimal:

Code:

```
// io/quasifix.cpp
string quasifix(double);
int main() {
  for ( auto x : { 1.5, 12.32,
      123.456, 1234.5678 } )
    cout << quasifix(x) << '\n';
```

Output:

```
   1.5
  12.32
 123.456
1234.5678
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

103. Scientific notation

Combining width and precision:

Code:

```
// io/iof.cpp
x = 1.234567;
cout << scientific;
for (int i=0; i<10; ++i) {
  cout << setw(10) << setprecision
      (4)
        << x << '\n';
  x *= 10;
}
cout << '\n';
```

Output:

```
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

104. Text output to file

Use:

Code:

```
// io/fio.cpp
#include <fstream>
using std::ofstream;
    /* ... */
  ofstream file_out;
  file_out.open
    ("fio_example.out");
    /* ... */
  file_out << number << '\n';
  file_out.close();
```

Output:

```
echo 24 | ./fio ; \
          cat
      ↪fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal 'file'.

105. String an object, 2

Redefine the less-less operator:

```
// geom/pointfunc.cpp
std::ostream& operator<<
(std::ostream &out,Point &p) {
  out << p.as_string(); return out;
};
```

Use:

```
// geom/pointfunc.cpp
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << "\n";
```

106. Formatter for your class

```
// iofmt/fmtclass.cpp
template<>
class std::formatter<XYclass>{
public:
  constexpr auto parse( std::format_parse_context& ctx ) {
    return ctx.begin(); };
  auto format( const XYclass& z,std::format_context& ctx ) const {
    return std::format_to
      ( ctx.out(),"[ {}, {} ]",z.x,z.y);
  };
};
```

Lambdas

107. Why lambda expressions?

Lambda expressions (sometimes incorrectly called ‘closures’) are ‘anonymous functions’. Why are they needed?

- Small functions may be needed; defining them is tedious, would be nice to just write the function recipe in-place.
- C++ can not define a function dynamically, depending on context.
  Example:
  1. we read `float c`
  2. now we want function `float f(float)` that multiplies by `c`:

```
float c; cin >> c;
float mult( float x ) { // DOES NOT WORK
  // multiply x by c
};
```

108. Introducing: lambda expressions

Traditional function usage:
explicitly define a function and apply it:

```
float sum(float x,float y) { return x+y; }
cout << sum( 1.2f, 3.4f );
```

New:
apply the function recipe directly:

Code:

```
// lambda/lambdaex.cpp
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Output:

```
3.8
```

109. Lambda syntax

```
[capture] ( inputs ) -> outtype { definition };
[capture] ( inputs ) { definition };
```

- The square brackets are how you recognize a lambda; we will get to the 'capture' later. For now it will often be empty.
- Inputs: like function parameters
- Result type specification `-> outtype`: can be omitted if compiler can deduce it;
- Definition: function body.

110. Direct invocation, 1

Invoke different constructors based on runtime condition:

Does not work:

```cpp
// lambda/invoke.cpp
if (foo)
  MyClass x(5,5);
else
  MyClass x("foo");
```

Solution:

```cpp
// lambda/invoke.cpp
auto x =
  [foo] () {
    if (foo)
      return MyClass(5,5);
    else
      return MyClass("foo");
  }();
```

Note the use of `auto` and the omitted return type.

111. Assign lambda expression to variable

Lambda expression assigned to a variable:

Code:

```
// lambda/lambdaex.cpp
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << '\n';
cout << summing ( 3.7, 5.2 ) << '\n';
```

Output:

```
3.8
8.9
```

- This is a variable declaration.
- Uses `auto` for technical reasons; see later.

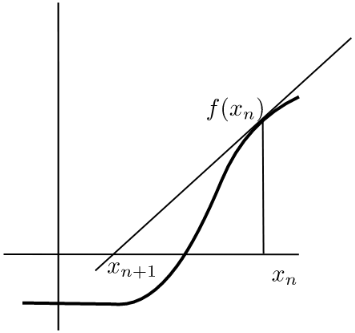
Return type could have been omitted:

```
auto summing =
[] (float x,float y) { return x+y; };
```

Example of lambda usage: Newton's method

112. Newton's method

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$



113. Newton for root finding

With

$$f(x) = x^2 - 2$$

zero finding is equivalent to

$$f(x) = 0 \quad \text{for } x = \sqrt{2}$$

so we can compute a square root if we have a zero-finding function.

Newton's method for this f:

$$x_{n+1} = x_n - f(x_n)/f'(x_n) = x_n - \frac{(x_n^2 - 2)}{2x_n} = x_n/2 + 2/x_n$$

Square root computation only takes division!

Exercise 29

Rewrite your code to use lambda functions for `f` and `fprime`.

If you use variables for the lambda expressions, put them in the main program.

You can base this off the file `newton.cpp` *in the repository*

114. Function pointers

You can pass a function to another function.
In C syntax:

```
void f(int i) { /* something with i */ };
void apply_to_5( (void)(*func)(int) ) {
    func(5);
}
int main() {
  apply_to_5(f);
}
```

(You don't have to understand this syntax. The point is that you can pass a function as argument.)

115. Lambdas as parameter

We want to write lambda expressions in-line in a function call:

```
int main() {
    apply_to_5
      ( [] (double x) { cout << x; } );
}
```

116. Lambdas as parameter: the problem

Lambdas have a type that is dynamically generated, so you can not write a function that takes a lambda as argument, because you can't write the type.

```
void apply_to_5( /* what? */ func ) {
    func(5);
}
int main() {
    apply_to_5
      ( [] (double x) { cout << x; } );
}
```

117. Lambdas as parameter: the solution

Header:

```
#include <functional>
using std::function;
```

declare function parameters by their signature
(that is, types of parameters and output):

Code:

```
// lambda/lambdaex.cpp
void apply_to_5
    ( function< void(int) > f ) {
  f(5);
}
    /* ... */
  apply_to_5
    ( [] (int i) {
      println("Int: {}",i);
     } );
```

Output:

```
Int: 5
```

118. Lambdas expressions for Newton

We are going to write a Newton function which takes two parameters: an objective function, and its derivative; it has a `double` as result.

```
// newton/newton-lambda.cpp
double newton_root
   ( function< double(double) > f,
     function< double(double) > fprime ) {
```

This states that *f*,*fprime* are in the class of `double(double)` functions: `double` parameter in, `double` result out.

Exercise 30

Rewrite the Newton exercise by implementing a `newton_root` function:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. then directly with the lambda expressions as arguments, that is, without assigning them to variables.

Captures

119. Capture variable

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

Code:

```
// lambda/lambdacapture.cpp
int n;
cin >> n;
auto increment_by_n =
  [n] ( int input ) -> int {
    return input+n;
};
println("{}",increment_by_n (5));
println("{}",increment_by_n (12));
println("{}",increment_by_n (25));
```

Output:

```
(input value:
      ↪1)

6
13
26
```

120. Capture more than one variable

Example: multiply by a fraction.

```
int d=2,n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

Exercise 31

- Set two variables

```
float low = .5, high = 1.5;
```

- Define a function `is_in_range` of one variable that tests whether that variable is between `low,high`.
  (Hint: what is the signature of that function? What is/are input parameter(s) and what is the return result?)

121. Capture value is copied

Illustrating that the capture variable is copied once and for all:

Code:

```
// lambda/lambdacapture.cpp
int inc;
cin >> inc;
auto increment =
  [inc] ( int input ) -> int {
    return input+inc;
  };
println("increment by: {}",inc);
println("1 -> {}",increment(1));
inc = 2*inc;
println("1 -> {}",increment(1));
```

Output:

```
increment by: 2
1 -> 3
1 -> 3
```

Exercise 32

Extend the Newton exercise to compute roots in a loop:

```
// newton/newton−lambda.cpp
  for ( float n=2.; n<10; n:=.5 ) {
    cout << "sqrt(" << n << ") = "
         << newton_root(
    /* ... */
                      )
         << '\n';
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

However, the *newton_root* function takes a function of only a real argument. Use a capture to make *f* dependent on the integer parameter.

122. Derivative by finite difference

You can approximate the derivative of a function f as

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

where h is small.
This is called a ‘finite difference’ approximation.

Exercise 33

Write a version of the root finding function that only takes the objective function:

```
double newton_root( function< double(double)> f )
```

and approximates the derivative by a finite difference. You can use a fixed value `h=1e-6`.

Do not reimplement the whole newton method: instead create a lambda expression for the gradient and pass it to the function `newton_root` you coded earlier as second argment.

123. Lambda in object

A set of integers, with a test on which ones can be admitted:

Class:

```
// lambda/lambdafun.cpp
#include <functional>
using std::function;
    /* ... */
class SelectedInts {
private:
  vector<int> bag;
  function< bool(int) > selector;
public:
  SelectedInts
      ( function< bool(int) > f ) {
    selector = f; };
```

Methods:

```
// lambda/lambdafun.cpp
  void add(int i) {
    if (selector(i))
      bag.push_back(i);
  };
  int size() {
    return bag.size(); };
  std::string as_string() {
    std::string s;
    for ( int i : bag )
      s += to_string(i)+" ";
    return s;
  };
};
```

124. Illustration

The above code in use:

Code:

```
// lambda/lambdafun.cpp
println("Give a divisor: ");
cin >> divisor;
println(".. divisor {}",divisor);
auto is_divisible =
  [divisor] (int i) -> bool {
    return i%divisor==0; };
SelectedInts multiples( is_divisible
      );
for (int i=1; i<50; ++i)
  multiples.add(i);
```

Output:

```
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

Advanced topics

125. Capture by value

Normal capture is by value:

Code:

```
// lambda/lambdacapture.cpp
int n;
cin >> n;
auto increment_by_n =
  [n] ( int input ) -> int {
    return input+n;
};
println("{}",increment_by_n (5));
println("{}",increment_by_n (12));
println("{}",increment_by_n (25));
```

Output:

```
(input value: 1)

6
13
26
```

126. Capture by reference

Capture a variable by reference so that you can update it:

Code:

```
// lambda/countif.cpp
int count=0;
auto count_if_f =
  [&count] (int i) {
    if (i%2==0) count++; };
for ( int i : {1,2,3,4,5} )
  count_if_f(i);
println("We counted: {}",count);
```

Output:

```
We counted: 2
```

127. Lambdas vs function pointers

Lambda expression with empty capture are compatible with C-style function pointers:

Code:

```cpp
// lambda/lambdacptr.cpp
int cfun_add1( int i ) {
  return i+1; };
int apply_to_5( int(*f)(int) ) {
  return f(5); };
    /* ... */
  auto lambda_add1 =
    [] (int i) { return i+1; };
  cout << "C ptr: "
       << apply_to_5(&cfun_add1)
       << '\n';
  cout << "Lambda: "
       << apply_to_5(lambda_add1)
       << '\n';
```

Output:

```
C ptr: 6
Lambda: 6
```

128. Use in algorithms

```
for_each( myarray, [] (int i) { cout << i; } );

transform( myarray, [] (int i) { return i+1; } );
```

See later.

Smart pointers

129. No more 'star' pointers

C pointers are barely needed.

- Use `std::string` instead of `char` array; use `std::vector` for other arrays.
- Parameter passing by reference: use actual references.
- Ownership of dynamically created objects: smart pointers.
- Pointer arithmetic: iterators.
- However: some legitimate uses later.

130. Smart pointers

Memory management is the whole point
so we only look at them in the context of objects.

Smart pointer: object with built-in reference counter:
counter zero ⇒ object can be freed.

131. Example: step 1, we need a class

Simple class that stores one number:

Definition:

```
// pointer/pointx.cpp
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto value() { return x; };
  void set(double xx) {
    x = xx; };
};
```

Example usage

```
// pointer/pointx.cpp
HasX xobj(5);
println("{}",xobj.value());
xobj.set(6);
println("{}",xobj.value());
```

132. Example: step 2, creating the pointer

Allocation of object and pointer to it in one:

```
auto X = make_shared<HasX>( /* args */ );

// or explicitly:

shared_ptr<HasX> X =
    make_shared<HasX>( /* constructor args */ );
```

133. Use of a shared pointer

Object vs pointed-object:

Code:

```
// pointer/pointx.cpp
#include <memory>
using std::make_shared;

    /* ... */
    HasX xobj(5);
    println("{}",xobj.value());
    xobj.set(6);
    println("{}",xobj.value());

    auto xptr = make_shared<HasX>(5);
    println("{}",xptr->value());
    xptr->set(6);
    println("{}",xptr->value());
```

Output:

```
5
6
5
6
```

134. Example: step 3: headers to include

Using smart pointers requires at the top of your file:

```
#include <memory>
using std::shared_ptr;
using std::make_shared;

using std::unique_ptr;
using std::make_unique;
```

135. Getting the underlying pointer

Demonstrate the `get` function:

Code:

```
// pointer/pointy.cpp
auto Y = make_shared<HasY>(5);
cout << Y->y << '\n';
Y.get()->y = 6;
cout << ( *Y.get() ).y << '\n';
```

Output:

```
5
6
```

136. Pointers don't go with addresses

The oldstyle `&y` address pointer can not be made smart:

```
// pointer/address.cpp
auto
  p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
     << p->y << '\n';
```

gives:

```
address(56325,0x7fff977cc380) malloc: *** error for object
    0x7ffeeb9caf08: pointer being freed was not allocated
```

137. Example: step 4: in use

Set two pointers to the same object:

Code:

```
// pointer/pointx.cpp
auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
println("{}",xptr->value());
yptr->set(6);
println("{}",xptr->value());
```

Output:

```
5
6
```

138. Pointer dereferencing

Example: function

```
float distance_to_origin( Point p );
```

How do you apply that to a `shared_ptr<Point>`?

```
shared_ptr<Point> p;
distance_to_origin( *p );
```

139. Null pointer

Initialize smart pointer to null pointer; test on null value:

```
shared_ptr<Foo> foo_ptr; // is nullptr by default initialization;
// stuff
if (foo_ptr!=nullptr)
  foo_ptr->do_something();
```

Exercise 34

With this code given:

Code:

```
// pointer/dynrectangle.cpp
float dx( Point other ) {
  return other.x-x; };
  /* ... */
  // main, with objects
  Point
    oneone(1,1), fivetwo(5,2);
  float dx = oneone.dx(fivetwo);
  /* ... */
  // main, with pointers
  auto
    oneonep  = make_shared<Point>(1,1),
    fivetwop = make_shared<Point>(5,2);
```

Output:

```
dx: 4
dx: 4
```

compute the `dx` between the `oneonep` & `fivetwop`.

You can base this off the file `dynrectangle.cpp` *in the repository*

Exercise 35

Make a `DynRectangle` class, which is constructed from two shared-pointers-to-`Point` objects:

```
// pointer/dynrectangle.cpp
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

Exercise 36

Test this design: Calculate the area, scale the top-right point, and recalculate the area:

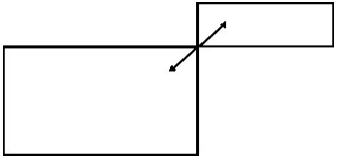
Code:

```
// pointer/dynrectangle.cpp
cout << "Area: " <<  lielow.area()
     << '\n';
/* ... */
cout << "Area: " <<  lielow.area()
     << '\n';
```

Output:

```
Area: 10
Area: 40
```

140. For the next exercise



Exercise 37

Make two `DynRectangle` objects so that the top-right corner of the first is the bottom-left corner of the other.

Now shift that point. Print out the two areas before and after to check correct behavior.

Automatic memory management

141. Memory leaks

C has a ‘memory leak’ problem

```
// the variable `array' doesn't exist
for ( /* lots of iterations */ ) {
  // attach memory to `array':
  double *array = new double[N];
  // do something with array;
  // forget to free
}
// the variable `array' does not exist anymore
// but the memory is still reserved.
```

The application ‘is leaking memory’.
Java/Python have ‘garbage collection’: runtime impact
C++ has the best solution: smart pointers with reference counting.

142. Illustration

We need a class with constructor and destructor tracing:

```
// pointer/ptr1.cpp
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

143. Constructor / destructor in action

Code:

```
// pointer/ptr0.cpp
cout << "Outside\n";
{
  thing x;
  cout << "create done\n";
}
cout << "back outside\n";
```

Output:

```
Outside
.. calling
      ↪constructor
create done
.. calling destructor
back outside
```

144. Illustration 1: pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
// pointer/ptr1.cpp
cout << "set pointer1"
     << '\n';
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << '\n';
thing_ptr1 = nullptr;
```

Output:

```
set pointer1
.. calling
      ↪constructor
overwrite pointer
.. calling destructor
```

145. Illustration 2: pointer copy

Code:

```
// pointer/ptr2.cpp
cout << "set pointer2" << '\n';
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << '\n';
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << '\n';
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << '\n';
thing_ptr3 = nullptr;
```

Output:

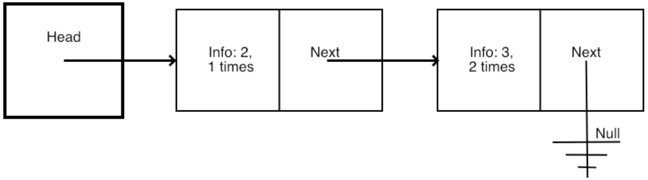
```
set pointer2
.. calling
     ↪constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

146. Reference counting

- The object counts how many pointers there are:
- ‘reference counting’
- A pointed-to object is deallocated if no one points to it.

Example: linked lists

147. Linked list



You can base this off the file `linkshared.cpp` in the repository

148. Definition of List class

A linked list has as its only member a pointer to a node:

```
// tree/linkshared.cpp
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

149. Definition of Node class

A node has information fields, and a link to another node:

```
// tree/linkshared.cpp
class Node {
private:
  int datavalue{0},datacount{0};
  shared_ptr<Node> next{nullptr};
public:
  Node() {};
  Node(int value,shared_ptr<Node> next=nullptr)
    : datavalue(value),datacount(1),next(next) {};
```

A Null pointer indicates the tail of the list.

150. List methods

List testing and modification.

```
List mylist;
cout << "Empty list has length: "
     << mylist.length() << '\n';

mylist.insert(3);
cout << "After one insertion the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
```

151. Recursive functions

- List structure is recursive
- Algorithms are naturally formulated recursively.

152. Recursive length computation

For the list:

```
// tree/linkshared.cpp
int List::length() {
  if (head==nullptr)
    return 0;
  else
    return head->length();
};
```

For a node:

```
// tree/linkshared.cpp
int Node::length() {
  if (!has_next())
    return 1;
  else
    return 1+next->length();
};
```

153. Iterative functions

- Recursive functions may have performance problems
- Iterative formulation possible

154. Iterative computation of the list length

Use a shared pointer to go down the list:

```
// tree/linkshared.cpp
int List::length_iterative() {
  int count = 0;
  if (head!=nullptr) {
    auto current_node = head;
    while (current_node->has_next()) {
      current_node = current_node->nextnode(); count += 1;
    }
  }
  return count;
};
```

(Fun exercise: can do an iterative de-allocate of the list?)

155. Print a list

Auxiliary function so that we can trace what we are doing.

Print the list head:

```
// tree/linkshared.cpp
void List::print() {
  cout << "List:";
  if (head!=nullptr)
    cout << " => "; head->print();
  cout << '\n';
};
```

Print a node and its tail:

```
// tree/linkshared.cpp
void Node::print() {
  cout << datavalue << ":" <<
     datacount;
  if (has_next()) {
    cout << ", "; next->print();
  }
};
```

156. Unique pointers

- Unique pointer: object can have only one pointer to it.
- Such a pointer can not be copied, only ‘moved’
  (that’s a whole other story)
- Potentially cheaper because no reference counting.

157. Definition of List class

A linked list has as its only member a pointer to a node:

```
// tree/linkunique.cpp
class List {
private:
  unique_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

158. Definition of Node class

A node has information fields, and a link to another node:

```
// tree/linkunique.cpp
class Node {
  friend class List;
private:
  int datavalue{0},datacount{0};
  unique_ptr<Node> next{nullptr};
public:
  friend class List;
  Node() {}
  Node(int value,unique_ptr<Node> tail=nullptr)
    : datavalue(value),datacount(1),next(move(tail)) {};
  ~Node() { cout << "deleting node " << datavalue << '\n'; };
```

A Null pointer indicates the tail of the list.

159. Iterative computation of the list length

Use a bare pointer, which is appropriate here because it doesn't own the node.

```
// tree/linkunique.cpp
int listlength() {
  Node *walker = next.get();
  int len = 1;
  while ( walker!=nullptr ) {
    walker = walker->next.get();
    ++len;
  }
  return len;
};
```

```
// tree/linkshared.cpp
int List::length_iterative() {
  int count = 0;
  if (head!=nullptr) {
    auto current_node = head;
    while (current_node->has_next()) {
      current_node = current_node->nex
    }
  }
  return count;
};
```

(You will get a compiler error if you try to make `walker` a smart pointer: you can not copy a unique pointer.)

160. Iterative vs bare pointers

- Use smart pointers for ownership
- Use bare pointers for pointing but not owning.
- This is an efficiency argument.
  I'm not totally convinced.

Union-like things

Tuples

161. Examples for this lecture

Example: two roots of a quadratic polynomial

Example: compute square root, or report that the input is negative

Example: roots of a quadratic polynomial: zero or one or two.

162. Returning two things

Simple solution:

```
// union/optroot.cpp
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = std::sqrt(x);
  return true;
};
```

Other solution: tuples

163. Tuple (de)construction

Headers

```
#include <tuple>
using std::tuple, std::make_tuple, std::pair, std::make_pair;
```

Construct:

```
tuple<int,char,double> icd;
auto icd = make_tuple(5,'d',3.14);
```

Deconstruct ('structured binding'):

```
auto [i,c,d] = icd;
// or by reference:
auto& [i,c,d] = icd;
```

164. Pairs

A `pair` is the same as a `tuple` of two elements:

```
pair<char,float> cf = make_pair('a',1.1f);
```

but there are two extra data members `first` `second`

```
auto [c,f] = cf;
// or
auto c=cf.first; auto f=cf.second;
```

165. Function returning tuple

How do you return two things of different types?

```
#include <tuple>
using std::make_tuple, std::tuple;

tuple<bool,float> maybe_root1(float x) {
  if (x<0)
    return make_tuple<bool,float>(false,-1);
  else
    return make_tuple<bool,float>(true,sqrt(x));
};

```

(not the best solution for the ‘root’ code)

166. Returning tuple with type deduction

Return type deduction:

```
// stl/tuple.cpp
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple
      <bool,float>
        (true,sqrt(x));
};
```

Alternative:

```
// stl/tuple.cpp
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

Note: use `pair` for `tuple` of two.

167. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
// stl/tuple.cpp
auto [succeed,y] = maybe_root2(x);
if (succeed)
  cout << "Root of " << x
       << " is " << y << '\n';
else
  cout << "Sorry, " << x
       << " is negative" << '\n';
```

Output:

```
Root of 2 is 1.41421
Sorry, -2 is negative
```

This is known as structured binding.

168. Exercises for the 'abc' formula

Introduce synonym:

```
// union/abctuple.cpp
using quadratic = tuple<double,double,double>;
    /* ... */
  // polynomial: x^2 - 2
  auto sunk = quadratic(1.,0.,-2);
```

You can base this off the file `abctuple.cpp` in the repository

169. Discriminant

Discriminant of the quadratic polynomial

Definition:

```
// union/abctuple.cpp
double discriminant
    ( quadratic q ) {
  auto [a,b,c] = q;
  return b*b-4*a*c;
}
```

Use:

```
// union/abctuple.cpp
auto d = discriminant( sunk );
cout << "discriminant: "
     << d << '\n';
```

Exercise 38

Write the function `abc_roots` that makes this code work:

```
// union/abctuple.cpp
auto roots = abc_roots( sunk );
auto [xplus,xminus] = roots;
cout << xplus << "," << xminus << '\n';
```

170. C++11 style tuples

Use `get` to unpack a tuple:

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
    // or:
    std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic = make_pair( 24, 'd' );
```

Annoyance: all that ‘get’ting.

Optional

171. Optional results

The most elegant solution to ‘a number or an error’ is to have a single quantity that you can query whether it’s valid.

```
#include <optional>
using std::optional;
```

```
// union/optroot.cpp
optional<float> MaybeRoot(float x) {
  if (x<0)
    return {};
  else
    return std::sqrt(x);
};
```

172. Testing and getting value

Two ways:

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} )
  if ( auto root = MaybeRoot(x);
       root.has_value() )
    cout << "Root is "
         << root.value()
         << '\n';
  else
    cout
      << "could not take root of "
      << x << '\n';
```

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} )
  if ( auto root = MaybeRoot(x) ;
     root )
    cout << "Root is "
         << *root << '\n';
  else
    cout
      << "could not take root of "
      << x << '\n';
```

Exercise 39

Continue the ‘abc’ exercise above and write a function that returns optionally a string,

```
// union/abctuple.cpp
optional<string> how_many_roots( quadratic q );
```

saying ‘one’ or ‘two’ if there are real roots:

```
// union/abctuple.cpp
auto num_solutions = how_many_roots(sunk);
if ( not num_solutions.has_value() )
  cout << "none\n";
else
  cout << num_solutions.value() << '\n';
```

Expected (C++23)

173. Expected

Expect double, return info string if not:

Function returning expected:

```
// union/expected.cpp
std::expected<double,string>
square_root( double x ) {
  if (x<0)
    return std::unexpected("
     negative");
  else if (x<limits<double>::min())
    return std::unexpected("
     underflow");
  else {
    auto result = sqrt(x);
    return result;
  }
}
```

In Use:

```
// union/expected.cpp
auto root = square_root(x);
if (x)
  cout << "Root=" << root.value()
     << '\n';
else if (root.error()==/* et cetera
     */ )
  /* handle the problem */
```

Variants

174. Variant

- Tuple of value and bool: we really need only one
- variant: it *is* one or the other
- You can set it to either, test which one it is.

175. Square root with variant

Return square root or false result:

```
// union/optroot.cpp
#include <variant>
using std::variant, std::get_if;
    /* ... */
variant<bool,float> RootVariant(
     float x) {
  if (x<0)
    return false;
  else
    return std::sqrt(x);
};
```

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} ) {
  auto okroot = RootVariant(x);
  auto root = get_if<float>(&okroot
     );
  if ( root )
    cout << "Root is "
         << *root << '\n';
  auto nope = get_if<bool>(&okroot)
     ;
  if (nope)
    cout
      << "could not take root of "
      << x << '\n';
}
```

176. More variant examples

Illustrating the usage:

```
// union/intdoublestring.cpp
variant<int,double,string> union_ids;
```

The `index` returns 0, 1, 2, and we get the value accordingly:

```
// union/intdoublestring.cpp
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << '\n';
}
```

177. Variant methods

```
// union/intdoublestring.cpp
variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

```
// union/intdoublestring.cpp
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << '\n';
}
```

```
// union/intdoublestring.cpp
union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
  cout << "Int: " << *union_int << '\n';
else if ( auto union_string = get_if<string>(&union_ids) ; union_string
    )
  cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

Exercise 40

Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

```
// union/abctuple.cpp
auto root_cases = abc_cases( sunk );
switch (root_cases.index()) {
case 0 : cout << "No roots\n"; break;
case 1 : cout << "Single root: " << get<1>(root_cases); break;
case 2 : {
  auto xs = get<2>(root_cases);
  auto [xp,xm] = xs;
  cout << "Roots: " << xp << "," << xm << '\n';
} ; break;
}
```

178. Monostate

How did you handle the case of no roots?
Possibility:

```
return std::monostate{};
```

'boolean with only one value'

Iterators

179. Range-based iteration

You have seen

```
for ( auto n : set_of_integers )
  if ( even(n) )
    do_something(n);
```

Can we do

```
for ( auto n : set_of_integers
    and even ) // <= not actual syntax
  do_something(n);
```

or even

```
// again, not actual syntax
apply( set_of_integers and even,
    do_something ):
```

180. Range algorithms

Algorithms: for-each, find, filter, transform ...

Ranges: iteratable things such as vectors

Views: transformations of ranges, such as picking only even numbers

181. Range over vector

With

```
// rangestd/range.cpp
vector<int> v{2,3,4,5,6,7};
```

Code:

```
// rangestd/range.cpp
#include <ranges>
namespace rng = std::ranges;
#include <algorithm>
    /* ... */
  vector<int> v{2,3,4,5,6,7};
  rng::for_each
    ( v, [] (int i) { cout << i << "
      "; } );
```

Output:

```
2 3 4 5 6 7
```

182. Range with accumulation

Capture a global accumulator by reference:

Code:

```
// rangestd/range.cpp
count = 0;
rng::for_each
  ( v,
    [&count] (int i) {
      count += (i<5); }
    );
cout << "Under five: "
     << count << '\n';
```

Output:

```
Under five: 3
```

Exercise 41

Revisit the vector normalization and rewrite the `norm` function to use a `for_each` algorithm.

Also rewrite the *`scale`* function using a `for_each` algorithm.

183. Range composition

Pipeline of ranges and views:

```
// rangestd/range.cpp
vector<int> v{2,3,4,5,6,7};
```

Code:

```
// rangestd/range.cpp
count = 0;
rng::for_each
  ( v | rng::views::drop(1),
    [&count] (int i) {
      count += (i<5); }
    );
cout << "minus first: "
     << count << '\n';
```

Output:

```
minus first: 2
```

'pipe operator'

184. Filter

Filter a range by some condition:

Code:

```cpp
// rangestd/filter.cpp
vector<float> numbers
  {1,-2.2,3.3,-5,7.7,-10};
for ( auto n :
        numbers
        | std::ranges::views::filter
          ( [] (float f) -> bool {
              return f>0; } )
    )
  cout << n << " ";
cout << '\n';
```

Output:

```
1 3.3 7.7
```

Exercise 42

Change the filter example to let the lambda count how many elements were > 0.

185. Range composition

Code:

```
// range/filtertransform.cpp
vector<int> v{ 1,2,3,4,5,6 };
/* ... */
auto times_two_over_five = v
  | rng::views::transform
      ( [] (int i) {
        return 2*i; } )
  | rng::views::filter
      ( [] (int i) {
        return i>5; } );
```

Output:

```
Original vector:
  1, 2, 3, 4, 5, 6,
Times two over five:
  6 8 10 12
```

186. Quantor-like algorithms

Code:

```
// rangestd/of.cpp
vector<int> integers{1,2,3,5,7,10};
auto any_even =
  std::ranges::any_of
    ( integers,
      [=] (int i) -> bool {
        return i%2==0; }
     );
if (any_even)
  cout << "there was an even\n";
else
  cout << "none were even\n";
```

Output:

```
there was an even
```

Also `all_of`, `none_of`

187. Reductions

`accumulate` and `reduce`:
tricky, and not in all compilers.
See above for an alternative.

188. Iota and take

Code:

```
// rangestd/iota.cpp
#include <ranges>
namespace rng = std::ranges;
    /* ... */
  for ( auto n :
          rng::views::iota(2,6) )
    cout << n << '\n';
  cout << "===\n";
  for ( auto n :
          rng::views::iota(2)
          | rng::views::take(4) )
    cout << n << '\n';
```

Output:

```
2
3
4
5
===
2
3
4
5
```

Exercise 43

A perfect number is the sum of its own divisors:

$$6 = 1 + 2 + 3$$

Output the perfect numbers.
(at least 4 of them)
Use only ranges and algorithms, no explicit loops.

1. Write a lambda expression to compute the sum of the factors of a number
2. Use `iota` to iterate over numbers: if one is equal to the sum of its factors, print it out.
3. Use *`filter`* to pick out these numbers.

Namespaces

189. What is the problem?

Name conflicts:

- there is the `std::vector`
- you want to write your own geometry library with a `vector` class
  $\Rightarrow$ conflict
- also unintentional conflicts from using multiple libraries

190. Solution: namespaces

A namespace is a ‘prefix’ for identifiers:

```
std::vector xstd; // standard namespace
geo::vector xgeo; // my geo namespace
lib::vector xlib; // from some library.
```

191. Namespaces in action

How do you indicate that something comes from a namespace?

Option: explicitly indicated.

```
#include <vector>
int main() {
  std::vector<stuff> foo;
}
```

Import the whole namespace:

```
#include <vector>
using namespace std;
int main() {
  vector<stuff> foo;
}
```

Good compromise:

```
#include <vector>
using std::vector;
int main() {
  vector<stuff> foo;
}
```

192. Defining a namespace

Introduce new namespace:

```
namespace geometry {
  // definitions
  class vector {
    // stuff
  };
```

193. Namespace usage

Double-colon notation for namespace and type:

```
geometry::vector myobject();
```

or

```
using geometry::vector;
vector myobject();
```

or even

```
using namespace geometry;
vector myobject();
```

194. Why not ‘using namespace std’?

Illustrating the dangers of `using namespace std`:

This compiles, but should not:

```
// func/swapname.cpp
#include <iostream>
using namespace std;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

This gives an error:

```
// func/swapusing.cpp
#include <iostream>
using std::cout;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

195. Guideline

- `using namespace` is ok in main program or implementation file
- Never! Ever! in a header file

Example

196. Example of using a namespace

Suppose we have a `geometry` namespace containing a `vector`, in addition to the `vector` in the standard namespace.

```
// namespace/geo.cpp
#include <vector>
#include "geolib.hpp" // this contains the geometry namespace
int main() {
  // std vector of geom segments:
  std::vector<geometry::segment> segments;
  segments.push_back( geometry::segment( geometry::point(1,1),geometry::
     point(4,5) ) );
```

What would the implementation of this be?

197. Namespace'd declarations

```
// namespace/geolib.hpp
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point( double x,double y );
    double dx(point);
    double dy(point);
  };
  class segment {
  private:
    point from,to;
```

198. Namespace'd implementations

```
// namespace/geolib.cpp
namespace geometry {
  point::point( double x,double y ) {
      xcoord = x; ycoord = y; };
  double point::dx( point other ) {
    return other.xcoord-xcoord; };
    /* ... */
  template< typename T >
  vector<T>::vector( std::string name,int size )
    : name_(name),std::vector<T>::vector(size) {};
}
```

Templates

199. Templated type name

If you have multiple functions or classes that do 'the same' for multiple types, you want the type name to be a variable, a template parameter. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...

// usually:
template <typename T>
```

200. Example: function

Definition:

```
// template/func.cpp
template <typename Real>
void sqrt_diff( Real x ) {
  cout << std::sqrt(x)-1.772 << '\n';
};
```

We use this with a templated function:

Code:

```
// template/func.cpp
sqrt_diff<float>( 3.14f );
sqrt_diff<double>( 3.14 );
```

Output:

```
4.48513e-06
4.51467e-06
```

Exercise 44

Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function *machine_eps* so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
// template/eps.cpp
float float_eps =
  machine_eps<float>();
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

double double_eps =
  machine_eps<double>();
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << '\n';
```

Output:

```
Epsilon float:
    ↪1.1921e-07
Epsilon double:
    ↪2.2204e-16
```

201. Templated vector

The templated vector class looks roughly like:

```
template<typename T>
class vector {
private:
  T *vectordata; // internal data
public:
  T at(int i) { return vectordata[i]  };
  int size() { /* return size of data */ };
  // much more
}
```

Exceptions

202. Throw an integer

Throw an integer error code:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

203. Catching an exception

Catch an integer:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

Exercise 45

Revisit the prime generator class (exercise 61) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 3.)

Code:

```
// primes/genx.cpp
try {
  do {
    auto cur = primes.nextprime();
    cout << cur << '\n';
  } while (true);
} catch ( string s ) {
  cout << s << '\n';
}
```

Output:

```
9931
9941
9949
9967
9973
Reached max int
```

204. Multiple catches

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
}
```

205. Catch any exception

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

206. Exception classes

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

207. Exceptions in constructors

A function try block will catch exceptions, including in member initializer lists of constructors.

```
B::B( int i )
  try : A(i) {
    // constructor body
  }
  catch (...) { // handle exception
  }
```

208. More about exceptions

Some remarks.

- Functions were able to define what exceptions they throw until C++17:

```
void func() throw( MyError, std::string );
```

- There are many predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section **??**.
- The main program acts as if an implicit `try/except` block is placed around it. You can replace the handler for that. See the exception header file.
- The keyword `noexcept` indicates that a function can not throw an exception:

```
void f() noexcept { ... };
```

This can be important in some applications such as embedded

209. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
// object/exceptdestruct.cpp
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor "
         << '\n'; };
  ~SomeObject() {
    cout <<  "calling the destructor "
         << '\n'; };
};
    /* ... */
  try {
    SomeObject obj;
    cout << "Inside the nested scope " << '\n';
    throw(1);
  } catch (...) {
    cout << "Exception caught" << '\n';
```

Output:

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

210. Using assertions

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some computation */;
}
```

Code design to facilitate testing the result:

```
float positive_outcome = /* some computation */
assert( positive_outcome>0 );
return positive_outcome;
```

211. Assertions to catch logic errors

Sanity check on things 'that you just know are true':

```
#include <cassert>
...
assert( bool expression )
```

Example:

```
x = sin(2.81);
y = x*x;
z = y * (1-y);
assert( z>=0. and z<=1. );
```

212. Use assertions during development

Assertions are disabled by

```
#define NDEBUG
```

before the include.

You can pass this as compiler flag:
`icpc -DNDEBUG yourprog.cpp`

Auto

213. Type deduction

In:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                  ( 20, new myclass(1.3) );
```

the compiler can figure it out:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
              ( 20, new myclass(1.3) );
auto result = someobject.somemethod();
```

214. Type deduction in functions

Function return type deduction:

```
// auto/autofun.cpp
auto equal(int i,int j) {
  return i==j;
};
```

215. Type deduction in methods

Return type of methods can be deduced:

```
// auto/plainget.cpp
class A {
private: float data;
public:
  A(float i) : data(i) {};
  auto &access() {
    return data; };
  void print() {
    cout << "data: " << data << '\n'; };
};
```

216. Auto and references, 1

Demostrating that `auto` discards references from the rhs:

Code:

```
// auto/plainget.cpp
A my_a(5.7);
// reminder: float& A::access()
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output:

```
data: 5.7
```

217. Auto and references, 2

Combine `auto` and references:

Code:

```
// auto/refget.cpp
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output:

```
data: 6.7
```

218. Auto and references, 3

For good measure:

```
// auto/constrefget.cpp
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1; // WRONG does not compile
my_a.print();
```

Casts

219. C++ casts

- `reinterpret_cast`: Old-style 'take this byte and pretend it is XYZ'; very dangerous.
  Instead:
- `bit_cast` (C++20): cast one object to another, with the exact same bit pattern.
- `static_cast`: simple scalar stuff
- `static_cast`: cast base to derived without check.
- `dynamic_cast:` cast base to derived with check.
- `const_cast`: Adding/removing const

Also: syntactically clearly recognizable.
no reason for using the old 'paren' cast

220. Static cast

```
// cast/longint.cpp
int hundredk = 100000;
int overflow =
  hundredk*hundredk;
cout << "overflow: "
     << overflow << '\n';
size_t bignumber =
  static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: "
     << bignumber << '\n';
```

Code:

```
// cast/intlong.cpp
long int hundredg = 100000000000;
cout << "long number:     "
     << hundredg << '\n';
int overflow;
overflow = static_cast<int>(hundredg
     );
cout << "assigned to int: "
     << overflow << '\n';
```

Output:

```
long number:
      ↪100000000000
assigned to int:
      ↪1215752192
```

221. Pointer to base class

Class and derived:

```
// cast/toderived.cpp
class Base {
public:
  virtual void print() = 0;
};
class Derived : public Base {
public:
  virtual void print() {
    cout << "Call Derived function!"
         << '\n'; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Call Erived function!"
         << '\n'; };
};
```

222. Cast to derived class

This is how to do it:

Code:

```
// cast/toderived.cpp
cout << "Dynamic cast to Derived\n";
Derived object1;
f(object1);
Erived object2;
f(object2);
```

Output:

```
Dynamic cast to
    ↪Derived
Call Derived
    ↪function!
Could not be cast to
    ↪Derived
```

223. Cast to derived class, the wrong way

Do not use this function g:

Code:

```
// cast/toderivedp.cpp
Base *object = new Derived();
g(object);
Base *nobject = new Erived();
g(nobject);
```

Output:

```
Static cast to
    ↪Derived
Call Derived
    ↪function!
Call Erived function!
```

Const

224. Why const?

- Clean coding: express your intentions whether quantities are supposed to not alter.
- Functional style programming: prevent side effects.
- NOT for optimization: the compiler does not use this for ‘constant hoisting’ (moving constant expression out of a loop).

225. Constant arguments

Function arguments marked `const` can not be altered by the function code. The following segment gives a compilation error:

```
// const/constchange.cpp
void f(const int i) {
  ++i; // COMPILER ERROR!
}
```

226. Const ref parameters

```
void f( const int &i ) { .... }
```

- Pass by reference: no copying, so cheap
- Const: no accidental altering.
- Especially useful for large objects.

227. No side-effects

It encourages a functional style, in the sense that it makes side-effects impossible:

```
class Things {
private:
  int i;
public:
  int getval() const { return i; }
  int inc() { return i++; } // side-effect!
  void addto(int &thru) const { // effect through parameter
    thru += i; }
}
```

228. Const polymorphism

Const and non-const version of `at`:

Code:

```cpp
// const/constat.cpp
class has_array {
private:
  vector<float> values;;
public:
  has_array(int l,float v)
    : values(vector<float>(l,v)) {};
  auto& at(int i) {
    cout << "var at" << '\n';
    return values.at(i); };
  const auto& at (int i) const {
    cout << "const at" << '\n';
    return values.at(i); };
  auto sum() const {
    float p;
    for ( int i=0; i<values.size(); ++i)
      p += at(i);
    return p;
  };
};
```

Output:

```
const at
const at
const at
1.5
var at
const at
const at
const at
4.5
```

Exercise 46

Explore variations on this example, and see which ones work and which ones not.

1. Remove the second definition of `at`. Can you explain the error?
2. Remove either of the `const` keywords from the second `at` method. What errors do you get?

229. Constexpr if

The combination `if constexpr` is useful with templates:

```
template <typename T>
auto get_value(T t) {
  if constexpr (std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

230. Constant functions

To declare a function to be constant, use `constexpr`. The standard example is:

```
constexpr double pi() {
  return 4.0 * atan(1.0); };
```

but also

```
constexpr int factor(int n) {
  return n <= 1 ? 1 : (n*fact(n-1));
}
```

(Recursion in C++11, loops and local variables in C++14.)

231. Mutable example

Code:

```
// object/mutable.cpp
class has_stuff {
private:
  mutable optional<complicated> thing = {};
public:
  const complicated& get_thing() const {
    if ( not thing.has_value() )
      thing = complicated(5);
    else cout << "thing already there\n";
    return thing.value();
  };
};
```

Output:

```
making complicated thing
thing already there
thing already there
```

More STL

Complex

Complex numbers

232. Complex numbers

```
#include <complex>

complex<float> f;
f.re = 1.; f.im = 2.;
complex<double> d(1.,3.);

using std::complex_literals::i;
std::complex<double> c = 1.0 + 1i;

conj(c); exp(c);
```

Complex Newton

Exercise 47

Rewrite your Newton code where the algorithm is in the main program so that it works for complex numbers. Here is the main; you need to write the functions:

```
// newton/newton-complex.cpp
complex<double> z{.5,.5};
while ( true ) {
  auto fz = f(z);
  cout << "f( " << z << " ) = " << fz << '\n';
  if (std::abs(fz)<1.e-10 ) break;
  z = z - fz/fprime(z);
}
```

You may run into the problem that you can not operate immediately between a complex number and a `float` or `double`. Use `static_cast`;

```
static_cast< complex<double> >(2)
```

Templated functions

233. Templatized Newton, first attempt

You can templatize the objective function and the derivative:

```cpp
// newton/newton-double.cpp
template<typename T>
T f(T x) { return x*x - 2; };
template<typename T>
T fprime(T x) { return 2 * x; };
```

and then write in the main program:

```cpp
// newton/newton-double.cpp
double x{1.};
while ( true ) {
  auto fx = f<double>(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime<double>(x);
}
```

Exercise 48

Update your *newton_root* function with a template parameter. Test it by having a main program that computes some roots in `float`, `double`, and `complex<double>`.

Exercise 49

Use your complex Newton method to compute $\sqrt{2}$. Does it work?

How about $\sqrt{-2}$? If it doesn't work, why, and can you fix this?

Exercise 50

Write a Newton method where the objective function is itself a template parameter, not just its arguments and return type. Hint: no changes to the main program are needed.

Then compute $\sqrt{-2}$ as:

```
// newton/lambda-complex.cpp
cout << "sqrt -2 = " <<
  newton_root<complex<double>>
  ( // objective function
    [] (complex<double> x) -> complex<double> {
        return x*x + static_cast<complex<double>>(2); },
    // derivative
    [] (complex<double> x) -> complex<double> {
        return x * static_cast<complex<double>>(2); },
    // initial value
    complex<double>{.1,.1}
    )
     << '\n';
```

Random

234. What are random numbers?

- Not really random, just very unpredictable.
- Often based on integer sequences:

$$r_{n+1} = ar_n + b \mod N$$

- $\Rightarrow$ they repeat, but only with a long period.
- A good generator passes statistical tests.
- ... a bad generator gives bad science (Ising model)

235. Random workflow

Use header:

```
#include <random>
```

Steps:

1. First there is the random engine which contains the mathematical random number generator.
2. The random numbers used in your code then come from applying a distribution to this engine.
3. Optionally, you can use a random seed, so that each program run generates a different sequence.

236. Random generators and distributions

- Random device

```
// default seed
std::default_random_engine generator;
// random seed:
std::random_device r;
std::default_random_engine generator{ r() };
```

- Distributions:

```
std::uniform_real_distribution<float> distribution(0.,1.);
std::uniform_int_distribution<int> distribution(1,6);
```

- Sample from the distribution:

```
std::default_random_engine generator;
std::uniform_int_distribution<> distribution(0,nbuckets-1);
random_number = distribution(generator);
```

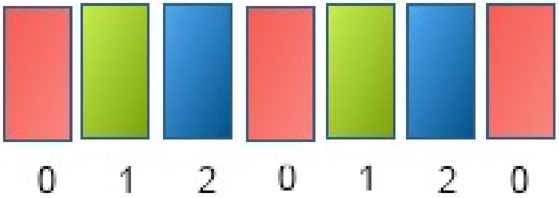
- Do not use the old C-style `random`!

237. Why so complicated?

- Large period wanted; C random has 2^{15} (implementation dependent)
- Multiple generators, guarantee on quality.
- Simple transforms have a bias:

```
int under100 = rand() % 100
```

Simple example: period 7, mod 3



238. Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

239. Poisson distribution

Poisson distributed integers:
chance of *k* occurrences, if *m* is the average number
(or $1/m$ the probability)

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

240. Local engine

Wrong approach: random generator local in the function.

Code:

```
// rand/static.cpp
int nonrandom_int(int max) {
  std::default_random_engine engine;
  std::uniform_int_distribution<>
    ints(1,max);
  return ints(engine);
};
    /* ... */
  // call 'nonrandom_int' three times
```

Output:

```
Three ints: 1, 1, 1.
```

Generator gets recreated in every function call.

Exercise 51

What is wrong with the following code:

```
int somewhat_random_int(int max) {
  random_device r;
  default_random_engine generator{ r() };
  std::uniform_int_distribution<> ints(1,max);
  return ints(generator);
};
```

241. Global engine

Good approach: random generator static in the function.

Code:

```
// rand/static.cpp
int realrandom_int(int max) {
  static  std::default_random_engine
      static_engine;
  std::uniform_int_distribution<>
    ints(1,max);
  return ints(static_engine);
};
```

Output:

```
Three ints: 15, 98,
     ↪70.
```

A single instance is ever created.

242. What does ‘static’ do?

- Static variable in function:
  persistent, shared between function calls
- Static variable in class:
  shared between all objects of that class

243. Class with static member

Class that counts how many objects have been generated:

Code:

```cpp
// object/static.cpp
class Thing {
private:
  static inline int nthings{0};
  int mynumber;
public:
  Thing() {
    mynumber = nthings++;
    cout << "I am thing "
         << mynumber << '\n';
  };
};
```

Output:

```
I am thing 0
I am thing 1
I am thing 2
```

(the `inline` is needed for the initialization)

Optional exercise 52

In the previous Goldbach exercise you had a prime number generator in a loop, meaning that primes got recalculated a number of times.

Optimize your prime number generator so that it remembers numbers already requested.

Hint: have a `static` vector.

244. Generator in a class

Note the use of `static`:

```cpp
// rand/randname.cpp
class generate {
private:
  static inline std::default_random_engine engine;
public:
  static int random_int(int max) {
    std::uniform_int_distribution<> ints(1,max);
    return ints(generate::engine);
  };
};
```

Usage:

```cpp
auto nonzero_percentage = generate::random_int(100)
```

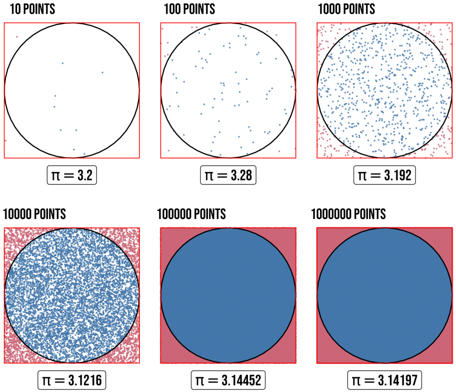
245. About seeding

- No seed: $\Rightarrow$ the same numbers every time
  … but not between different compilers / computers
- Explicit seed: reproducible.
- Average result ('ensembles'): use many different seeds.
- Seeding in parallel is tricky.

Integration

246. Compute pi by Monte Carlo method

- Generate many random coordinates $(x, y) \in [0, 1]^2$.
- Count the ratio of inside-the-circle to total.
- Compute π from that.



Exercise 53

Code this.

How many samples does it take to get $2, 3, 4, \ldots$ digits accuracy?

247. Volume of ball

The surface and volume of an n-dimensional ball satisfy the recurrences:

$$V_n = S_{n-1}/n; \quad S_n = 2\pi V_{n-1} \qquad \text{where } V_0 = 1,\ V_1 = 2.$$

```
// rand/nball.cpp
realtype pi;
realtype surface( int d );
realtype volume( int d ) {
  if (d==0) return 1;
  else if (d==1) return 2;
  else
    return surface(d-1)/d;
};
realtype surface( int d ) {
  if (d==0) return 2;
  else
    return 2 * pi * volume(d-1);
};
```

Exercise 54

Compute the volume V_n by generating random n-dimensional coordinates, and counting wether they are in the unit ball.

Limits

248. Templated functions for limits

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;
```

249. Limits of floating point values

Limits of floating point numbers:

Code:

```
// stl/limits.cpp
println("Single lowest {}",
    numeric_limits<float>::lowest())
    ;
println(" .. and epsilon {}",
    numeric_limits<float>::epsilon()
    );
println("Double lowest {}",
    numeric_limits<double>::lowest()
    );
println(" .. and epsilon {}",
    numeric_limits<double>::epsilon
    ());
```

Output:

```
Single lowest
    ↪-3.4028235e+38
 .. and epsilon
    ↪1.1920929e-07
Double lowest
    ↪-1.7976931348623157e+308
 .. and epsilon
    ↪2.220446049250313e-16
```

Random numbers

250. Random generators and distributions

- Random device

```
// default seed
std::default_random_engine generator;
// random seed:
std::random_device r;
std::default_random_engine generator{ r() };
```

- Distributions:

```
std::uniform_real_distribution<float> distribution(0.,1.);
std::uniform_int_distribution<int> distribution(1,6);
```

- Sample from the distribution:

```
std::default_random_engine generator;
std::uniform_int_distribution<> distribution(0,nbuckets-1);
random_number = distribution(generator);
```

- Do not use the old C-style `random`!

251. Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

Time

252. Chrono

```
#include <chrono>

// several clocks
using myclock = std::chrono::high_resolution_clock;

// time and duration
auto start_time = myclock::now();
auto duration = myclock::now()-start_time;
auto microsec_duration =
    std::chrono::duration_cast<std::chrono::microseconds>
                (duration);
cout << "This took "
     << microsec_duration.count() << "usec\n"
```

253. Date

coming in C++20

File system

254. file system

```
#include <filesystem>
```

including directory walker

Regular expressions

255. Regular expressions

Code:

```
// regexp/regsearch.cpp
  {
    auto findthe = regex("the");
    auto found = regex_search
      ( sentence,findthe );
    assert( found==true );
    cout << "Found <<the>>" << '\n';
  }
  {
    smatch match;
    auto findthx = regex("o[^o]+o");
    auto found = regex_search
      ( sentence, match ,findthx );
    assert( found==true );
    cout << "Found <<o[^o]+o>>"
         << " at " << match.position
     (0)
         << " as <<" << match.str(0)
      << ">>"
         << " preceeded by <<" << 
     match.prefix() << ">>"
         << '\n';
  }
```

Output:

```
Found <<the>>
Found <<o[^o]+o>> at
    ↪12 as <<own
    ↪fo>> preceeded
    ↪by <<The quick
    ↪br>>
```

C++20 modules

256. Modules

Sorry, I don't have a compiler yet that allows me to test this.

Unit testing

Intro to testing

257. Dijkstra quote

> *Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)*

Still ...

258. Types of testing

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

259. Unit testing

- Every part of a program should be testable
- $\Rightarrow$ good idea to have a function for each bit of functionality
- Positive tests: show that code works when it should
- Negative tests: show that the code fails when it should

260. Unit testing

- Every part of a program should be testable
- Do not write the tests after the program:
  write tests while you develop the program.
- Test-driven development:
  1. design functionality
  2. write test
  3. write code that makes the test work

261. Principles of TDD

Develop code and tests hand-in-hand:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

262. Unit testing frameworks

Testing is important, so there is much software to assist you.

Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

Intro to Catch2

263. Toy example

Function and tester:

```
// catch/require.cpp
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

int five() { return 5; }

TEST_CASE( "needs to be 5" ) {
    REQUIRE( five()==5 );
}
```

The define line supplies a main:
you don't have to write one.

264. Tests that fail

```
// catch/require.cpp
float fiveish() { return 5.00001; }
TEST_CASE( "not six" ) {
  // this will fail
  REQUIRE( fivish()==5 );
  // this will succeed
  REQUIRE( fivish()==Catch::Approx(5) );
}
```

Exercise 55

Write a boolean function *`is_prime`*, and write a test case for it. This should have both cases that succeed and that fail.

265. Boolean tests

Test a boolean expression:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Compound boolean expressions need to be parenthesized:

```
REQUIRE( ( x>0 and x<1 ) );
```

266. Output for failing tests

Run the tester:

Code:

```
// catch/false.cpp
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

int five() { return 6; }

TEST_CASE( "needs to be 5" ) {
    REQUIRE( five()==5 );
}
```

Output:

```
-------------------------
needs to be 5
-------------------------
false.cpp:21
.........................

false.cpp:22: FAILED:
  REQUIRE( five()==5
    ↪)
with expansion:
  6 == 5

=========================
test cases: 1 | 1
    ↪failed
assertions: 1 | 1
    ↪failed
```

267. Diagnostic information for failing tests

`INFO`: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "iteration: " << n ); // only printed for failed tests
    REQUIRE( f(n)>0 );
}
```

268. Exceptions

Exceptions are a mechanism for reporting an error:

```
double SquareRoot( double x ) {
  if (x<0) throw(1);
  return std::sqrt(x);
};
```

More about exceptions later;
for now: Catch2 can deal with them

269. Test for exceptions

Suppose a function `g(n)` satisfies:

- it succeeds for input $n > 0$
- it fails for input $n \leq 0$:
  throws exception

```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

270. Slightly realistic example

We want a function that

- computes a square root for $x \geq 0$
- throws an exception for $x < 0$;

```
// catch/sqrt.cpp
double root(double x) {
  if (x<0) throw(1);
  return std::sqrt(x);
};

TEST_CASE( "test sqrt function" ) {
  double x=3.1415, y;
  REQUIRE_NOTHROW( y=root(x) );
  REQUIRE( y*y==Catch::Approx(x) );
  REQUIRE_THROWS( y=root( -3.14 ) );
}
```

What happens if you require:

```
REQUIRE( y*y==x );
```

271. Tests with code in common

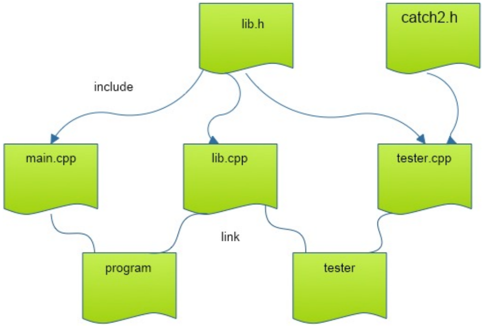
Use `SECTION` if tests have intro/outtro in common:

```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

Catch2 file structure

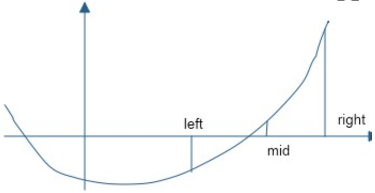
272. Realistic setup

- All program functionality in a ‘library’:
  split between header and implementation
- Main program can be short
- Tester file with only tests.
- (Tester also needs the `catch2` stuff included)



TDD example: Bisection

273. Root finding by bisection



- Start with bounds where the function has opposite signs.

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

- Find the mid point;
- Adjust either left or right bound.

274. Coefficient handling

$$f(x) = c_0 x^d + c_1 x^{d-1} \cdots + c_{d-1} x^1 + c_d$$

We implement this by constructing a `polynomial` object from coefficients in a `vector<double>`:

```
// bisect/zeroclasslib.hpp
class polynomial {
private:
  std::vector<double> coefficients;
public:
  polynomial( std::vector<double> c );
```

Exercise 56: Test for proper coefficients

For polynomial coefficients to give a well-defined polynomial, the zero-th coefficient needs to be non-zero:

```
// bisect/zeroclasstest.cpp
TEST_CASE( "proper test","[2]" ) {
  vector<double> coefficients{3., 2.5, 2.1};
  REQUIRE_NOTHROW( polynomial(coefficients) );

  coefficients.at(0) = 0.;
  REQUIRE_THROWS( polynomial(coefficients) );
}
```

Write a constructor that accepts the coefficients, and throws an exception if the above condition is violated.

275. Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
For this exercise we reject even degree polynomials.

```
// bisect/zeroclassmain.cpp
if ( not third_degree.is_odd() ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

This test will be used later;
first we need to implement it.

Exercise 57: Odd degree testing

Implement the *is_odd* test.

Gain confidence by unit testing:

```
// bisect/testzeroarray.cpp
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

276. Test on polynomials evaluation

Next we need to evaluate polynomials.

Equality testing on floating point is dangerous:
use *Catch*::*Approx*(*sb*)

```
// bisect/zeroclasstest.cpp
polynomial second( {2,0,1.1} );
// correct interpretation: 2x^2 + 1.1
REQUIRE( second.evaluate_at(2) == Catch::Approx(9.1) );
// wrong interpretation: 1.1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6.4) );
polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
REQUIRE( third(0) == Catch::Approx(1) );
```

Exercise 58: Evaluation, looking neat

Make polynomial evaluation work, but use overloaded evaluation:

```
// bisect/zeroclasstest.cpp
polynomial second( {2,0,1.1} );
// correct interpretation: 2x^2 + 1.1
REQUIRE( second(2) == Catch::Approx(9.1) );
polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
REQUIRE( third(0) == Catch::Approx(1) );
```

277. Finding initial bounds

We need a function *`find_initial_bounds`* which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

(can you write that more compactly?)

Exercise 59: Test for initial bounds

In the test for proper initial bounds, we reject even degree polynomials and left/right points that are reversed:

```
// bisect/zeroclasstest.cpp
double left{10},right{11};
right = left+1;
polynomial second( {2,0,1} ); // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
double
  leftval = third(left),
  rightval = third(right);
REQUIRE( leftval*rightval<=0 );
```

Can you add a unit test on the left/right values?

278. Move the bounds closer

Root finding iteratively moves the initial bounds closer together:

```
// bisect/zeroclasslib.hpp
void move_bounds_closer
    ( const polynomial&,double& left,double& right,bool trace=false );
```

- on input, `left`<`right`, and
- on output the same must hold.
- … but the bounds must be closer together.
- Also: catch various errors
- Also also: optional trace parameter; you leave that unusued.

Exercise 60: Test moving bounds

```
// bisect/zeroclasstest.cpp
  REQUIRE_THROWS( move_bounds_closer(third,right,left) );
  REQUIRE_THROWS( move_bounds_closer(third,left,left) );

  double old_left = left, old_right = right;
  REQUIRE_NOTHROW( move_bounds_closer(third,left,right) );
  leftval = third(left); rightval = third(right);
  REQUIRE( leftval*rightval<=0 );
  REQUIRE( ( ( left==old_left and right<old_right ) or
      ( right==old_right and left>old_left ) ) );
```

279. Putting it all together

Ultimately we need a top level function

```
double find_zero( polynomial coefficients,double prec );
```

- reject even degree polynomials
- set initial bounds
- move bounds closer until close enough: $|f(y)| < \mathtt{prec}$.

Exercise 61: Put it all together

Make this call work:

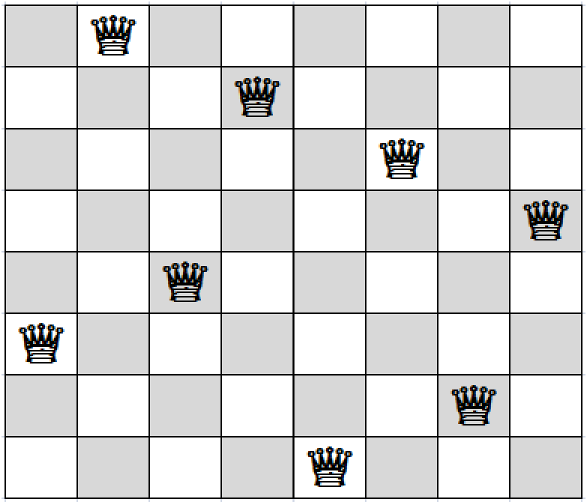
```
// bisect/zeroclassmain.cpp
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

TDD example: Eight queens

280. Classic problem

Can you put 8 queens on a board so that they can't hit each other?



281. Statement

- Put eight pieces on an 8×8 board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

282. Not good solution

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

Better: abort search early.

Exercise 62: Board class

Class *board*:

```
// queens/queens.hpp
ChessBoard(int n);
```

Method to keep track how far we are:

```
// queens/queens.hpp
int next_row_to_be_filled()
```

Test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 63: Place one queen

Method to place the next queen,
without testing for feasibility:

```
// queens/queens.hpp
void place_next_queen_at_column(int i);
```

This test should catch incorrect indexing:

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Without this test, would you be able to cheat?

Exercise 64: Test if we're still good

Feasibility test:

```
// queens/queens.hpp
bool feasible()
```

Some simple cases:
(add to previous test)

```
// queens/queentest.cpp
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
// queens/queentest.cpp
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

Exercise 65: Test collisions

```
// queens/queentest.cpp
ChessBoard collide = one;
// place a queen in a `colliding' location
collide.place_next_queen_at_column(0);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Exercise 66: Test a full board

Construct full solution

```
// queens/queens.hpp
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Test:

```
// queens/queentest.cpp
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 67: Exhaustive testing

This should now work:

```
// queens/queentest.cpp
// loop over all possibilities first queen
auto firstcol = GENERATE_COPY( range(1,n) );
ChessBoard place_one = empty;
REQUIRE_NOTHROW( place_one.place_next_queen_at_column(firstcol) );
REQUIRE( place_one.feasible() );

// loop over all possbilities second queen
auto secondcol = GENERATE_COPY( range(1,n) );
ChessBoard place_two = place_one;
REQUIRE_NOTHROW( place_two.place_next_queen_at_column(secondcol) );
if (secondcol<firstcol-1 or secondcol>firstcol+1) {
  REQUIRE( place_two.feasible() );
} else {
  REQUIRE( not place_two.feasible() );
}
```

Exercise 68: Place if possible

You need to write a recursive function:

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

- place the next queen.
- if stuck, return 'nope'.
- if feasible, recurse.

```
class board {
  /* stuff */
  optional<board> place_queens() const {
    /* stuff */
    board next(*this);
    /* stuff */
    return next;
  };
```

Exercise 69: Test last step

Test *place_queens* on a board that is almost complete:

```
// queens/queentest.cpp
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Note the new constructor! (Can you write a unit test for it?)

Exercise 70: Sanity tests

```
// queens/queentest.cpp
TEST_CASE( "no 2x2 solutions","[8]" ) {
  ChessBoard two(2);
  auto solution = two.place_queens();
  REQUIRE( not solution.has_value() );
}
```

```
// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

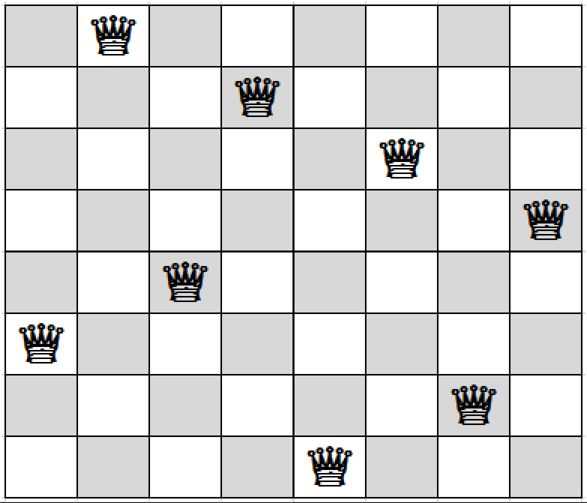
Exercise 71: O

ptional: can you do timing the solution time as function of the size of the board?

Eight queens problem by TDD (using objects)

283. Problem statement

Can you place eight queens on a chess board so that no pair threatens each other?

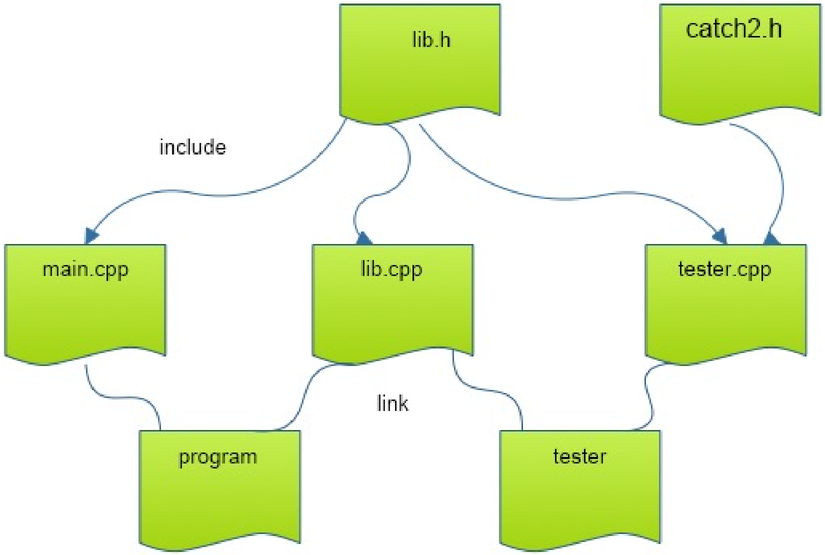


284. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

285. File structure



286. Basic object design

Object constructor of an empty board:

```
// queens/queens.hpp
ChessBoard(int n);
```

Test how far we are:

```
// queens/queens.hpp
int next_row_to_be_filled()
```

First test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 72: Board object

Start writing the `board` class, and make it pass the above test.

Exercise 73: Board method

Write a method for placing a queen on the next row,

```
// queens/queens.hpp
void place_next_queen_at_column(int i);
```

and make it pass this test (put this in a `TEST_CASE`):

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Exercise 74: Test for collisions

Write a method that tests if a board is collision-free:

```
// queens/queens.hpp
bool feasible()
```

This test has to work for simple cases to begin with. You can add these lines to the above tests:

```
// queens/queentest.cpp
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
// queens/queentest.cpp
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

```
// queens/queentest.cpp
ChessBoard collide = one;
// place a queen in a `colliding' location
collide.place_next_queen_at_column(0);
```

Exercise 75: Test full solutions

Make a second constructor to ‘create’ solutions:

```
// queens/queens.hpp
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Now we test small solutions:

```
// queens/queentest.cpp
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 76: No more delay: the hard stuff!

Write a function that takes a partial board, and places the next queen:

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

Test that the last step works:

```
// queens/queentest.cpp
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Alternative to using `optional`:

```
bool place_queen( const board& current, board &next );
// true if possible, false is not
```

Exercise 77: Test that you can find solutions

Test that there are no 3×3 solutions:

```
// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

but 4×4 solutions do exist:

```
// queens/queentest.cpp
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

History of C++ standards

287. C++98/C++03

Of the C++03 standard we only highlight deprecated features.

- `auto_ptr` was an early attempt at smart pointers. It is deprecated, and C++17 compilers will actually issue an error on it.

288. C++11

- `auto`
- Range-based for.
- Lambdas.
- Variadic templates.
- Smart pointers.
- `constexpr`

289. C++14

C++14 can be considered a bug fix on C++11. It simplifies a number of things and makes them more elegant.

- Auto return type deduction:
- Generic lambdas (section **??**) Also more sophisticated capture expressions.

290. C++17

- Optional; section **??**.
- Structured binding declarations as an easier way of dissecting tuples; section 229.
- Init statement in conditionals; section **??**.

291. C++20

- modules: these offer a better interface specification than using *header files*.
- coroutines, another form of parallelism.
- concepts including in the standard library via ranges; section **??**.
- spaceship operator including in the standard library
- broad use of normal C++ for direct compile-time programming, without resorting to template metaprogramming (see last trip reports)
- ranges
- calendars and time zones
- text formatting
- `span`. See section **??**.

292. C++23

- *md_span*