C++ for C Programmers

Victor Eijkhout

TACC Training 2024

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 (sometimes C++23) standard.

```
icpc    -std=c++20 yourprogram.cxx
g++     -std=c++20 yourprogram.cxx
clang++ -std=c++20 yourprogram.cxx
```

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

```
alias icpc='icpc -std=c++20'
et cetera
```

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_20 )
```

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.

7. What is not modern C++?

Do not use:

- Parameter passing with `&`
- *`malloc`* and such

It's legal, just not 'modern', and frankly not needed.

Minor enhancements

8. 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.

9. 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`)

10. 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)

11. Hello world!

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

Ok for now:

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

Better:

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

12. Hello world again!

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

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

13. Screen output

Many ways ...

```
#include <cstdio>
#include <iostream>
#include <iomanip>
#include <format>
using namespace std;
int main() {
  float pi=3.1416f;
  printf("pi is %5.3f\n",pi);
  cout << "pi is " << setprecision(4) << pi << '\n';
  cout << format("pi is {:.4}\n",pi);
  // C++23
  //print("pi is {:.4}\n",pi);
}
```

14. Small note

For convenience I may do

```
using namespace std;
```

but this is not generally advisable.
More later.

15. C standard header files

Equivalent of C `math.h` and such:

```
#include <cmath>
#include <cstdlib>
```

But a number of headers are not needed anymore / replaced by better.

Functions

16. Big and small changes

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

Parameter passing

17. Mathematical type function

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function. (Cost of copying?)
- pass by value
- 'functional programming'

18. Results other than through return

Also good design:

- Return no function result,
- or return return status (0 is success, nonzero various informative statuses), and
- return other information by changing the parameters.
- *pass by reference*
- Parameters are sometimes classified 'input', 'output', 'throughput'.

Later: `optional`, *`expected`*.

19. 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)

20. Reference

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

Code:

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

Output:

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

(You will not use references often this way.)

21. 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'
```

22. Reference vs pointer

- There are no 'null' references.
  (There is a `nullptr`, but that 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.

23. Parameter passing by reference

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

```
1  void f(int &n) {
2    n = /* some expression */ ;
3  };
4  int main() {
5    int i;
6    f(i);
7    // i now has the value that was set in the function
8  }
```

Reference syntax is cleaner than C 'pass by reference'

24. Pass by reference example 1

Code:

```
1 // basic/setbyref.cpp
2 void f( int &i ) {
3   i = 5;
4 }
5 int main() {
6
7   int var = 0;
8   f(var);
9   cout << var << '\n';
```

Output:

```
5
```

Compare the difference with leaving out the reference.

25. Pass by reference example 2

```
bool can_read_value( int &value ) {
  // this uses functions defined elsewhere
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status==0;
}

int main() {
  int n;
  if (!can_read_value(n)) {
    // if you can't read the value, set a default
    n = 10;
  }
  ..... do something with 'n' ....
```

26. 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:

```
1 // func/swap.cpp
2 int i=1,j=2;
3 cout << i << "," << j << '\n';
4 swap(i,j);
5 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:

```
1 // func/divisible.cpp
2 cout << number;
3 if (is_divisible
4       (number,divisor,remainder))
5   cout << " is divisible by ";
6 else
7   cout << " has remainder "
8        << remainder << " from ";
9 cout << divisor << '\n';
```

Output:

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

More about functions

27. 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.

28. 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
```

29. 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
```

Object-Oriented Programming

Classes

30. 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.

31. Running example

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

```
1 class Point {
2     /* stuff */
3 };
4 int main () {
5   Point p; /* stuff */
6 }
```

Exercise 3

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

32. Object functionality

Small illustration: point objects.

Code:

```
1 // object/functionality.cpp
2 Point p(1.,2.);
3 cout << "distance to origin "
4      << p.distance_to_origin()
5      << '\n';
6 p.scaleby(2.);
7 cout << "distance to origin "
8      << p.distance_to_origin()
9      << '\n'
10     << "and angle " << p.angle()
11     << '\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?

33. 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( /* ... */ );
```

34. 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);
```

```
1  class Point {
2  private: // data members
3    double x,y;
4  public: // function members
5    Point
6      (double x_in,double y_in){
7        x = x_in; y = y_in;
8    };
9    /* ... */
10 };
```

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

35. 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

36. Class methods

Definition and use of the *distance* function:

Code:

```
1 // geom/pointclass.cpp
2 class Point {
3 private:
4   float x,y;
5 public:
6   Point(float in_x,float in_y) {
7     x = in_x; y = in_y; };
8   float distance_to_origin() {
9     return sqrt( x*x + y*y );
10  };
11 };
12    /* ... */
13  Point p1(1.0,1.0);
14  float d = p1.distance_to_origin();
15  cout << "Distance to origin: "
16       << d << '\n';
```

Output:

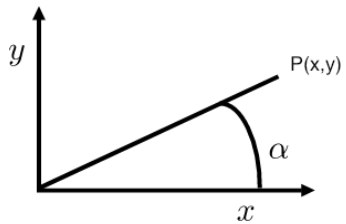
```
Distance to origin:
     1.41421
```

37. 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.

38. 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:

```
1 #include <cmath>
2 class Point {
3 private: // members
4   double r,theta;
5 public: // methods
6   Point( double x,double y ) {
7     r = sqrt(x*x+y*y);
8     theta = atan2(y/x);
9   }
```

Note: no change to outward API.

Exercise 7

Discuss the pros and cons of this design:

```
1  class Point {
2  private:
3    double x,y,r,theta;
4  public:
5    Point(double xx,double yy) {
6      x = xx; y = yy;
7      r = // sqrt something
8      theta = // something trig
9    };
10   double angle() { return alpha; };
11 };
```

39. Data access in methods

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

```
1 class Point {
2 private:
3   double x,y;
4 public:
5  void flip() {
6     Point flipped;
7     flipped.x = y; flipped.y = x;
8     // more
9   };
10 };
```

(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.

Review quiz 1

T/F?

- A class is primarily determined by the data it stores.
  `/poll "Class determined by its data" "T" "F"`
- A class is primarily determined by its methods.
  `/poll "Class determined by its methods" "T" "F"`
- If you change the design of the class data, you need to change the constructor call.
  `/poll "Change data, change constructor proto too" "T" "F"`

40. Methods that alter the object

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

Code:

```
1 // geom/pointscaleby.cpp
2 class Point {
3     /* ... */
4   void scaleby( double a ) {
5     x *= a; y *= a; };
6     /* ... */
7 };
8     /* ... */
9   Point p1(1.,2.);
10  cout << "p1 to origin "
11       << p1.length() << '\n';
12  p1.scaleby(2.);
13  cout << "p1 to origin "
14       << p1.length() << '\n';
```

Output:

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

Data initialization

41. 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.

42. Data initialization

The naive way:

```
1 class Point {
2 private:
3   double x,y;
4 public:
5   Point( double in_x,
6          double in_y ) {
7     x = in_x; y = in_y;
8   };
```

The preferred way:

```
1 // geom/pointinit.cpp
2 class Point {
3 private:
4   double x,y;
5 public:
6   Point( double in_x,
7          double in_y )
8     : x(in_x),y(in_y) {
9   }
```

Explanation later. It's technical.

Interaction between objects

43. Methods that create a new object

Code:

```
1 // geom/pointscale.cpp
2 class Point {
3     /* ... */
4   Point scale( double a ) {
5     auto scaledpoint =
6         Point( x*a, y*a );
7     return scaledpoint;
8   };
9     /* ... */
10  cout << "p1 to origin "
11       << p1.dist_to_origin()
12       << '\n';
13  Point p2 = p1.scale(2.);
14  cout << "p2 to origin "
15       << p2.dist_to_origin()
16       << '\n';
```

Output:

```
p1 to origin 2.23607
p2 to origin 4.47214
```

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

44. 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:

```
1 // geom/pointscale.cpp
2 Point Point::scale( double a )
      {
3   Point scaledpoint =
4     Point( x*a, y*a );
5   return scaledpoint;
6 };
```

Creates point, copies it to *new_point*

Better:

```
1 // geom/pointscale.cpp
2 Point Point::scale( double a )
      {
3   return Point( x*a, y*a );
4 };
```

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 *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?

45. Constructor/destructor

Constructor: function that gets called when you create an object.

```
MyClass {
public:
  MyClass( /* args */ ) { /* construction */ }
  /* more */
};
```

If you don't define it, you get a default.

Destructor (rarely used):
function that gets called when the object goes away, for instance when you leave a scope.

46. Using the default constructor

No constructor explicitly defined;

You recognize the default constructor in the main by the fact that an object is defined without any parameters.

Code:

```
1  // object/default.cpp
2  class IamOne {
3  private:
4    int i=1;
5  public:
6    void print() {
7      cout << i << '\n';
8    };
9  };
10     /* ... */
11   IamOne one;
12   one.print();
```

Output:

```
1
```

47. Default constructor

Refer to *Point* definition above.
Consider this code that looks like variable declaration, but for objects:

```
Point p1(1.5, 2.3);
Point p2;
p2 = p1.scaleby(3.1);
```

Compiling gives an error (g++; different for intel):

```
pointdefault.cpp: In function 'int main()':
pointdefault.cpp:32:21: error: no matching function for call to
                  'Point::Point()'
```

48. Default constructor

The problem is with *p2*:

```
Point p1(1.5, 2.3);
Point p2;
```

- *p1* is created with your explicitly given constructor;
- *p2* uses the default constructor:

  ```
  Point() {};
  ```

- default constructor is there by default, unless you define another constructor.
- you can redefine the default constructor:

  ```
  // geom/pointdefault.cpp
  Point() {};
  Point( double x,double y )
    : x(x),y(y) {};
  ```

  (but often you can avoid needing it)

Exercise 10

Make a class `LinearFunction` with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

49. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
1 // object/stream.cpp
2 class Stream {
3 private:
4   int last_result{0};
5 public:
6   int next() {
7     return last_result++; };
8 };
9
10 int main() {
11   Stream ints;
12   cout << "Next: "
13        << ints.next() << '\n';
14   cout << "Next: "
15        << ints.next() << '\n';
16   cout << "Next: "
17        << ints.next() << '\n';
```

Output:

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

50. 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

The Goldbach conjecture says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this 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.

51. 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:

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

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 relations: has-a

52. Has-a relationship

A class usually contains data members. These can be simple types or other classes. This allows you to reflect relations between things you are modeling.

```
1 class Person {
2   string name;
3   ....
4 };
5 class Course {
6 private:
7   Person the_instructor;
8   int year;
9 };`
```

This is called the has-a relation:

Course has-a *Person*

53. Literal and figurative has-a

A line segment has a starting point and an end point. *LineSegment* code design:

Store both points:

```
1 class Segment {
2 private:
3   Point p_start,p_end;
4 public:
5   Point end_point() {
6     return p_end; };
7 }
8 int main() {
9   Segment seg;
10  Point somepoint =
11    seg.end_point();
```

or store one and derive the other:

```
1 class Segment {
2 private:
3   Point starting_point;
4   float length,angle;
5 public:
6   Point end_point() {
7     /* some computation
8        from the
9        starting point */ };
10 }
```

Implementation vs API: implementation can be very different from user interface.

54. Polymorphism in constructors

You have to decide what to store and what to derive, but you can construct two ways:

```
1 class Segment {
2 private:
3  // up to you how to implement!
4 public:
5   Segment( Point start,float length,float angle )
6     { .... }
7   Segment( Point start,Point end ) { ... }
```

Advantage: with a good API you can change your mind about the implementation without changing the calling code.

Exercise 14

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point botleft,float width,float height);
   ```

   The logical implementation is to store these quantities. Implement methods:

   ```
   float area(); float rightedge_x(); float topedge_y();
   ```

   and write a main program to test these.

2. Add a second constructor

   ```
   Rectangle(Point botleft,Point topright);
   ```

   Can you figure out how to use member initializer lists for the constructors?

Class inheritance: is-a

55. 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*.

56. 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

```
1 class General {
2 protected: // note!
3  int g;
4 public:
5  void general_method() {};
6 };
7
8 class Special : public General {
9 public:
10   void special_method() { g = ... };
11 };
```

57. Inheritance: derived classes

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

```
1 int main() {
2   Special special_object;
3   special_object.general_method();
4   special_object.special_method();
5 }
```

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

58. Constructors

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

```
1 class General {
2 public:
3   General( double x,double y ) {};
4 };
5 class Special : public General {
6 public:
7   Special( double x ) : General(x,x+1) {};
8 };
```

59. 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 15

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?

60. Overriding methods

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

```
1 class Base {
2 public:
3   virtual f() { ... };
4 };
5 class Deriv : public Base {
6 public:
7   virtual f() override { ... };
8 };
```

61. Override and base method

Code:

```
1 // object/virtual.cpp
2 class Base {
3 protected:
4   int i;
5 public:
6   Base(int i) : i(i) {};
7   virtual int value() { return i; };
8 };
9
10 class Deriv : public Base {
11 public:
12   Deriv(int i) : Base(i) {};
13   virtual int value() override {
14     int ivalue = Base::value();
15     return ivalue*ivalue;
16   };
17 };
```

Output:

```
25
```

62. Friend classes

A friend class can access private data and methods even if there is no inheritance relationship.

```
1 /* forward definition: */ class A;
2 class B {
3   // A objects can access B internals:
4   friend class A;
5 private:
6   int i;
7 };
8 class  A {
9 public:
10  void f(B b) { b.i; }; // friend access
11 };
```

63. Abstract classes

Special syntax for virtual method:

```
1 class Base {
2 public:
3   virtual void f() = 0;
4 };
5 class Deriv : public Base {
6 public:
7   virtual void f() { ... };
8 };
```

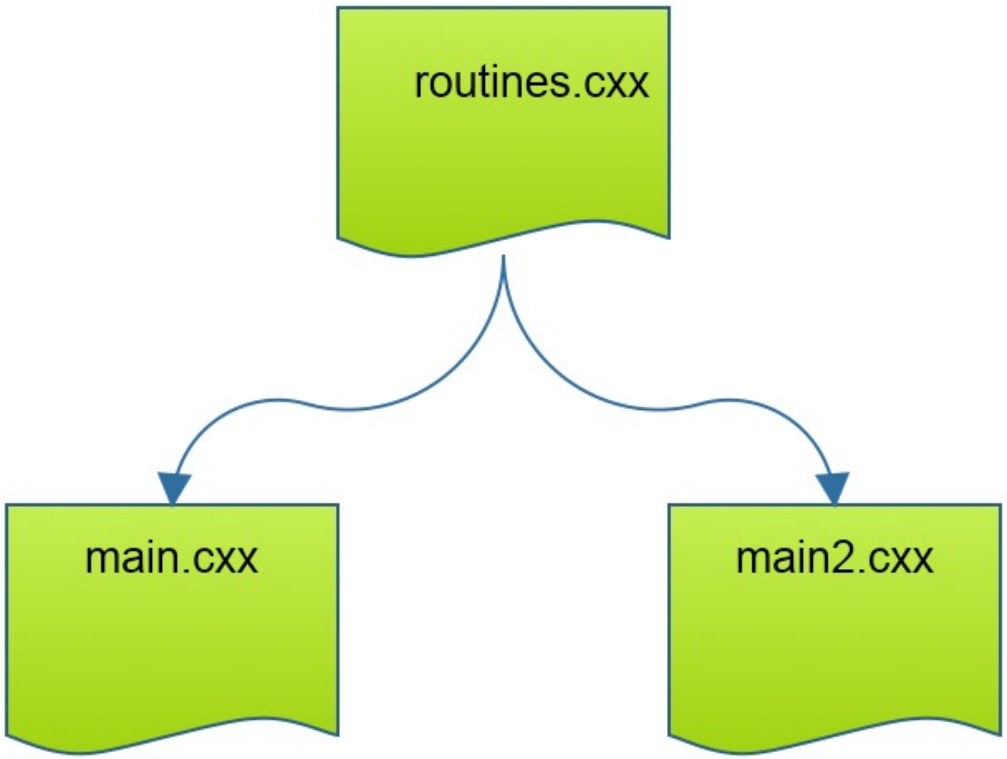
You can not create objects of the base class, only of the derived class.

64. More

- Multiple inheritance: an X is-a A, but also is-a B.
  This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.

65. Include files

- Code reuse is good.
- How would you use functions/classes in more than one main?



We will discuss systematic solutions.

66. Reminder: definition vs declaration

Definition:

```
bool iseven( int n ) { return n%2==0 }
```

Declaration:

```
bool iseven( int n );
// or even:
bool iseven( int );
```

67. Declarations, case 1

Some people like defining functions after the main.
Problem: the main needs to know about them.

'forward declaration'

```
int f(int);
int main() {
  f(5);
};
int f(int i) {
  return i;
}
```

versus:

```
int f(int i) {
  return i;
}
int main() {
  f(5);
};
```

This is a stylistic choice.

68. Declarations, case 2

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

69. Separate compilation

Split your program in multiple files.

- Easier to edit
- Less chance of `git` conflicts
- Only recompile the file you edit
  $\Rightarrow$ reduction of compile/build time.

70. Declarations for separate compilation

Define a function in one file;
an other file uses it, so needs the declaration:

```
// file: def.cpp
int tester(float x) {
  .....
}
```

```
// file : main.cpp
int tester(float);

int main() {
  int t = tester(...);
  return 0;
}
```

This Is Not A Good Design!

71. Declarations and header files

Using a header file with function declarations.

Header file contains only declaration:

```
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```
// file: def.cpp
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cpp
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

What happens if you leave out the `#include "def.h"` in both cases?

72. Class declarations

Header file:

```
// proto/functheader.hpp
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file: **missing snippet classheaderimpl**

73. File naming convention

- Source files: `.cpp` `.cxx`
  I use `.cpp` for no real reason
- Header files: `.h` `.hpp` `.hxx`
  I use `.hpp` by analogy with `.cpp`
  `.h` reminds me too much of C.

74. Compiling and linking

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile

   ```
   icpc -c yourfile.cc
   ```

   which gives you a file `yourfile.o`, a so-called object file; and
2. Then you use the compiler as linker to give you the executable file:

   ```
   icpc -o yourprogram yourfile.o
   ```

75. Dealing with multiple files

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

76. Header file with include guard

Header file tests if it has already been included:

```
// this is foo.h
#ifndef FOO_H
#define FOO_H

// the things that you want to include

#endif
```

This prevents double or recursive inclusion.

77. Make

Good idea to learn the `Make` utility for project management.

(Also `Cmake`.)

78. Skeleton example

Directory `skeletons/funct_skeleton` contains

```
funct.cpp functheader.hpp functmain.cpp
```

CMake setup:

```
add_executable(
    funct functmain.cpp funct.cpp functheader.hpp )
```

79. CMake compilation

Do `cmake` and then `make`:

```
[ 33%] Building CXX object CMakeFiles/funct.dir/functmain.cpp.o
[ 66%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[100%] Linking CXX executable funct
[100%] Built target funct
```

80. Justification for separate compilation

- Edit only `funct.cpp`;
- Do *not* `cmake`;
- do `make`

```
( cd build && make )
Consolidate compiler generated dependencies of target funct
[ 33%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[ 66%] Linking CXX executable funct
[100%] Built target funct
```

Only that file got recompiled.

81. Data members in proto

Data members, even private ones, need to be in the header file:

```
1 class something {
2 private:
3   int localvar;
4 public:
5   // declaration:
6   double somedo(vector);
7 };
```

Implementation file:

```
1 // definition
2 double something::somedo(vector v) {
3    .... something with v ....
4    .... something with localvar ....
5 };
```

Vectors

82. 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.
(Not even `new`.)

83. 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)
```

84. Short vectors

Short vectors can be created by enumerating their elements:

```
1  // array/shortvector.cpp
2  #include <vector>
3  using std::vector;
4
5  int main() {
6    vector<int> evens{0,2,4,6,8};
7    vector<float> halves = {0.5, 1.5, 2.5};
8    auto halfloats = {0.5f, 1.5f, 2.5f};
9    cout << evens.at(0)
10        << " from " << evens.size()
11        << '\n';
12   return 0;
13 }
```

Exercise 16

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

85. 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.

86. 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
```

87. Range over elements

Finding the maximum element

Code:

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

Output:

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

Exercise 17

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 18

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.

88. Range over vector denotation

Code:

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

Output:

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

89. Vector definition

Definition and/or initialization:

```
1 #include <vector>
2 using std::vector;
3
4 vector<type> name;
5 vector<type> name(size);
6 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.

90. Accessing vector elements

Square bracket notation (zero-based):

Code:

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

Output:

```
4,8
```

With bound checking:

Code:

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

Output:

```
4,8
```

Safer, slower.

91. Vector elements out of bounds

Square bracket notation:

Code:

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

Output:

```
1,4
```

With bound checking:

Code:

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

Output:

```
libc++abi: terminating
     with uncaught
     exception of type
     std::out_of_range:
     vector
```

Safer, slower.

92. 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:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2) << '\n';
```

Output:

```
6.6
```

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

93. Example: multiplying elements

Example: multiply all elements by two:

Code:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2) << '\n';
```

Output:

```
6.6
```

94. 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:

```
1 // array/vectoridxmax.cpp
2 int tmp_idx = 0;
3 int tmp_max = numbers.at(tmp_idx);
4 for (int i=0; i<numbers.size(); ++i) {
5   int v = numbers.at(i);
6   if (v>tmp_max) {
7     tmp_max = v; tmp_idx = i;
8   }
9 }
10 cout << "Max: " << tmp_max
11      << " at index: " << tmp_idx << '\n';
```

Output:

```
Max: 6.6 at
     index: 3
```

95. 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 19

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

Which mechanism do you use?

Exercise 20

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?

96. Vector copy

Vectors can be copied just like other datatypes:

Code:

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

Output:

```
3.5,7
```

Note: contents copied, not just pointer.

97. 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`.
- With iterators (see later): `insert`, `erase`

98. 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

99. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

```
1 // array/vectorend.cpp
2 vector<int> mydata(5,2);
3 mydata.push_back(35);
4 cout << mydata.size()
5      << '\n';
6 cout << mydata.back()
7      << '\n';
```

Output:

```
6
35
```

Similar functions: `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

100. 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.)

101. 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;
```

102. 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.

103. 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
```

Vectors and functions

104. 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:

```
1 // array/vectorreturn.cpp
2 vector<int> make_vector(int n) {
3   vector<int> x(n);
4   x.at(0) = n;
5   return x;
6 }
7     /* ... */
8   vector<int> x1 = make_vector(10);
9   // "auto" also possible!
10  cout << "x1 size: " << x1.size()
       << '\n';
11  cout << "zero element check: " <<
       x1.at(0) << '\n';
```

Output:

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

105. 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.

106. Vector pass by value example

Code:

```
1 // array/vectorpassnot.cpp
2 void set0
3   ( vector<float> v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v.at(0) << '\n';
```

Output:

```
3.5
```

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

107. Vector pass by reference

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

Code:

```
1 // array/vectorpassref.cpp
2 void set0
3   ( vector<float> &v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v.at(0) << '\n';
```

Output:

```
4.6
```

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

108. 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 21

Revisit exercise 20 and introduce a function for computing the L_2 norm.

(hints for the next exercise)

Random numbers:

```
// high up in your code:
#include <random>
using std::rand;

// in your main or function:
float r = 1.*rand()/RAND_MAX;
// gives random between 0 and 1
```

(You will learn a better random later)

Exercise 22

Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
vector<float> sorted = sort(values);
```

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place (overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)

Vectors in classes

109. Can you make a class around a vector?

You may want a class of objects that contain a vector. For instance, you may want to name your vectors.

```
1 class named_field {
2 private:
3   vector<double> values;
4   string name;
```

The problem here is when and how that vector is going to be created.

110. Create the contained vector

Use initializers for creating the contained vector:

```
class named_field {
private:
  string name;
  vector<double> values;
public:
  named_field( string name,int n )
    : name(name),
      values(vector<double>(n)) {
  };
};
```

Even shorter:

```
  named_field( string name,int n )
    : name(name),values(n) {
  };
```

Multi-dimensional arrays

111. Multi-dimensional vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

Create a row vector, then store 10 copies of that:
vector of vectors.

112. Matrix class

```
1  // array/matrix.cpp
2  class matrix {
3  private:
4    vector<vector<double>> elements;
5  public:
6    matrix(int m,int n)
7      : elements(
8                  vector<vector<double>>(m,vector<double>(n))
9                  ) {
10   }
11   void set(int i,int j,double v) {
12     elements.at(i).at(j) = v;
13   };
14   double get(int i,int j) {
15     return elements.at(i).at(j);
16   };
```

(Can you combine the *get/set* methods, using **???**)

Exercise 23

Write *rows*() and *cols*() methods for this class that return the number of rows and columns respectively.

Exercise 24

Write a method `void set(double)` that sets all matrix elements to the same value.

Write a method `double totalsum()` that returns the sum of all elements.

Code:

```
1 // array/matrix.cpp
2 A.set(3.);
3 cout << "Sum of elements: "
4      << A.totalsum() << '\n';
```

Output:

```
Sum of elements: 30
```

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

113. Matrix class; better design

Better idea:

```
// array/matrixclass.cpp
class matrix {
private:
  vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n)
    : m(m),n(n),the_matrix(m*n) {};
  void set(int i,int j,double v) {
    the_matrix.at( i*n +j ) = v;
  };
  double get(int i,int j) {
    return the_matrix.at( i*n +j );
  };
    /* ... */
};
```

(Old-style solution: use cpp macro)

Exercise 25

In the matrix class of the previous slide, why are `m,n` stored explicitly, unlike in the matrix class of section **??**?

Exercise 26

Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

114. Pascal's triangle

Pascal's triangle contains binomial coefficients:

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

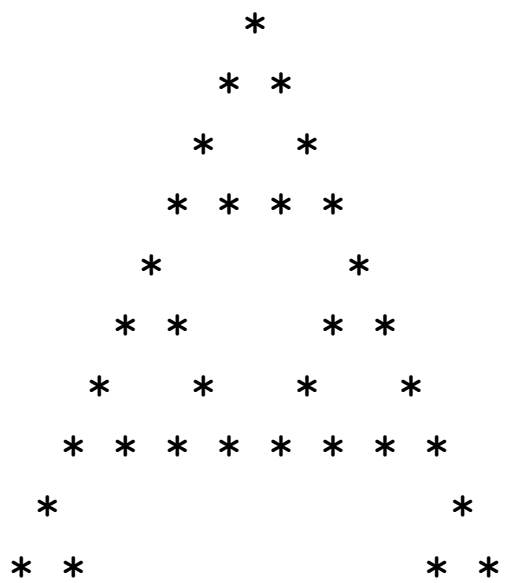
The coefficients can be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} & \end{cases}$$

(There are other formulas. Why are they less preferable?)

Exercise 27

- Write a class `pascal` so that `pascal(n)` is the object containing n rows of the above coefficients. Write a method `get(i,j)` that returns the (i, j) coefficient.
- Write a method `print` that prints the above display.
- First print out the whole pascal triangle; then:
- Write a method `print(int m)` that prints a star if the coefficient modulo m is nonzero, and a space otherwise.



115. Exercise continued

- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
- Your program should accept:
  1. an integer for the size
  2. any number of integers for the modulo; if this is zero, stop, otherwise print stars as described above.

Optional exercise 28

Extend the Pascal exercise:
Optimize your code to use precisely enough space for the coefficients.

Other array stuff

116. Array class

Static arrays:

```
#include <array>
std::array<int,5> fiveints;
```

- Size known at compile time.
- Vector methods that do not affect storage
- Zero overhead.

117. Random walk exercise

```
// rand/walk_lib_vec.cpp
class Mosquito {
private:
  vector<float> pos;
public:
  Mosquito( int d )
    : pos( vector<float>(d,0.f) ) { };

// rand/walk_lib_vec.cpp
void step() {
  int d = pos.size();
  auto incr = random_step(d);
  for (int id=0; id<d; ++id)
    pos.at(id) += incr.at(id);
};
```

Finish the implementation. Do you get improvement from using the array class?

118. Span

Create a `span` from a `vector`:

```
#include <span>
vector<double> v;
auto v_span = std::span<double>( v.data(),v.size() );
```

Strings

119. String declaration

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

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

(Do not use the C legacy mechanisms.)

120. 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";
```

121. Quotes in strings

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

Code:

```
1 // string/quote.cpp
2 string
3   one("a b c"),
4   two("a \"b\" c"),
5   three( R"("a ""b """c)" );
6 cout << one << '\n';
7 cout << two << '\n';
8 cout << three << '\n';
```

Output:

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

122. Concatenation

Strings can be *concatenated*:

Code:

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

Output:

```
foo bar: 7
```

123. String indexing

You can query the *size*:

Code:

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

Output:

```
5
```

or use subscripts:

Code:

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

Output:

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

124. Ranging over a string

Same as ranging over vectors.

Range-based for:

Code:

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

Output:

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

Ranging by index:

Code:

```
1 // string/stringrange.cpp
2 string abc = "abc";
3 cout << "By character: ";
4 for (int ic=0; ic<abc.size(); ic++)
5   cout << abc[ic] << " ";
6 cout << '\n';
```

Output:

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

125. Range with reference

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

Code:

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

Output:

```
Shifted: bcd
```

Review quiz 2

True or false?

1. `'0'` is a valid value for a `char` variable
   ```
   /poll "single-quote 0 is a valid char" "T" "F"
   ```
2. `"0"` is a valid value for a `char` variable
   ```
   /poll "double-quote 0 is a valid char" "T" "F"
   ```
3. `"0"` is a valid value for a `string` variable
   ```
   /poll "double-quote 0 is a valid string" "T" "F"
   ```
4. `'a'+'b'` is a valid value for a `char` variable
   ```
   /poll "adding single-quote chars is a valid char" "T" "F"
   ```

Exercise 29

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:

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

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

126. More vector methods

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

Code:

```
1 // string/strings.cpp
2 string five_chars;
3 cout << five_chars.size() << '\n';
4 for (int i=0; i<5; ++i)
5   five_chars.push_back(' ');
6 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 30

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 31

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

127. 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.

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

128. String an object, 1

Define a function that yields a string representing the object, and

```
1 // geom/pointfunc.cpp
2   string as_string() {
3     stringstream ss;
4     ss << "(" << x << "," << y << ")";
5     return ss.str();
6   };
7     /* ... */
8 std::ostream& operator<<
9 (std::ostream &out,Point &p) {
10   out << p.as_string(); return out;
11 };
```

129. String an object, 2

Redefine the less-less operator to use this.

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

Exercise 32

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

Code:

```
1 // io/quasifix.cpp
2 string quasifix(double);
3 int main() {
4   for ( auto x : { 1.5, 12.32,
      123.456, 1234.5678 } )
5     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.

I/O

130. Default unformatted output

Code:

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

Output:

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

131. Fancy formatting

1. Header: `iomanip`: manipulation of `cout`
2. std::format: looks like `printf` and Python's `print(f"stuff")`. Note: C++20 but not yet implemented; we use `fmtlib` instead.

132. Fmtlib

- Repo: `https://github.com/fmtlib/fmt.git`
- Compile flag:

  `-I${TACC_FMTLIB_INC`
- Link flag:

  `-L${TACC_FMTLIB_LIB} -lfmt`
- Include line:

  `#include <fmt/format.h>`

133. Fmtlib basics

- *print* for printing (C++23)
  format gives `std::string`;
- Arguments indicated by curly braces;
- braces can contain numbers (and modifiers, see next)

Code:

```
1 // io/fmtbasic.cpp
2 cout << format("{}\n",2);
3 string hello_string = format
4   ("{} {}!","Hello","world");
5 cout << hello_string << '\n';
6 print
7   ("{0}, {0}
      {1}!\n","Hello","world");
```

Output:

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

134. Reserve space

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

Code:

```
1  // io/width.cpp
2  #include <iomanip>
3  using std::setw;
4      /* ... */
5    cout << "Width is 6:" << '\n';
6    for (int i=1; i<200000000; i*=10)
7      cout << "Number: "
8           << setw(6) << i << '\n';
9    cout << '\n';
10
11   // `setw' applies only once:
12   cout << "Width is 6:" << '\n';
13   cout << ">"
14        << setw(6) << 1 << 2 << 3 <<
        '\n';
15   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
```

135. Right aligned in fmtlib

Code:

```
1 // io/fmtlib.cpp
2 for (int i=10; i<2000000000; i*=10)
3   fmt::print("{:>6}\n",i);
```

Output:

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

136. Padding character

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

Code:

```
1 // io/formatpad.cpp
2 #include <iomanip>
3 using std::setfill;
4 using std::setw;
5     /* ... */
6   for (int i=1; i<200000000; i*=10)
7     cout << "Number: "
8          << setfill('.')
9          << setw(6) << i
10         << '\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.

137. Left alignment

Instead of right alignment you can do left:

Code:

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

Output:

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

138. Padding characters in fmtlib

Code:

```
1 // io/fmtlib.cpp
2 for (int i=10; i<2000000000; i*=10)
3   fmt::print("{0:.>6}\n",i);
```

Output:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

139. Number base

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

Code:

```
 1 // io/format16.cpp
 2 #include <iomanip>
 3 using std::setbase;
 4 using std::setfill;
 5     /* ... */
 6   cout << setbase(16)
 7        << setfill(' ');
 8   for (int i=0; i<16; ++i) {
 9     for (int j=0; j<16; ++j)
10       cout << i*16+j << " " ;
11     cout << '\n';
12   }
```

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 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
```

140. Number bases in fmtlib

Code:

```
1 // io/fmtlib.cpp
2 fmt::print
3   ("{0} = {0:b} bin\n",17);
4 fmt::print
5   ("    = {0:o} oct\n",17);
6 fmt::print
7   ("    = {0:x} hex\n",17);
```

Output:

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

Exercise 33

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
```

141. Fixed point precision

Fixed precision applies to fractional part:

Code:

```
1 // io/fix.cpp
2 x = 1.234567;
3 cout << fixed;
4 for (int i=0; i<10; ++i) {
5   cout << setprecision(4) << x <<
      '\n';
6   x *= 10;
7 }
```

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 34

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

Code:

```
1 // io/quasifix.cpp
2 string quasifix(double);
3 int main() {
4   for ( auto x : { 1.5, 12.32,
      123.456, 1234.5678 } )
5     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.

142. Scientific notation

Combining width and precision:

Code:

```
1 // io/iof.cpp
2 x = 1.234567;
3 cout << scientific;
4 for (int i=0; i<10; ++i) {
5   cout << setw(10) <<
       setprecision(4)
6        << x << '\n';
7   x *= 10;
8 }
9 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
```

143. Text output to file

Use:

Code:

```
1 // io/fio.cpp
2 #include <fstream>
3 using std::ofstream;
4     /* ... */
5   ofstream file_out;
6   file_out.open
7     ("fio_example.out");
8     /* ... */
9   file_out << number << '\n';
10  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'.

144. Redefine less-less

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
// stl/ostream.cpp
class container {
    /* ... */
  int value() const {
    /* ... */
  };
    /* ... */
ostream &operator<<(ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
    /* ... */
  container eye(5);
  cout << eye << '\n';
```

Smart pointers

145. 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.

146. 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 $\Rightarrow$ object can be freed.

147. 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);
cout << xobj.value() << '\n';
xobj.set(6);
cout << xobj.value() << '\n';
```

148. 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 */ );
```

149. Use of a shared pointer

Object vs pointed-object:

Code:

```
1  // pointer/pointx.cpp
2  #include <memory>
3  using std::make_shared;
4
5      /* ... */
6      HasX xobj(5);
7      cout << xobj.value() << '\n';
8      xobj.set(6);
9      cout << xobj.value() << '\n';
10
11     auto xptr =
        make_shared<HasX>(5);
12     cout << xptr->value() << '\n';
13     xptr->set(6);
14     cout << xptr->value() << '\n';
```

Output:

```
5
6
5
6
```

150. 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;
```

151. Getting the underlying pointer

```
X->y;
// is the same as
X.get()->y;
// is the same as
( *X.get() ).y;
```

Code:

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

Output:

```
5
6
```

152. 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
```

153. Example: step 4: in use

Why do we use pointers?
Pointers make it possible for two variables to own the same object.

Code:

```
1 // pointer/pointx.cpp
2 auto xptr = make_shared<HasX>(5);
3 auto yptr = xptr;
4 cout << xptr->value() << '\n';
5 yptr->set(6);
6 cout << xptr->value() << '\n';
```

Output:

```
5
6
```

154. 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 );
```

155. Null pointer

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

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

Exercise 35

With this code given:

Code:

```
1 // pointer/dynrectangle.cpp
2 float dx( Point other ) {
3   return other.x-x; };
4   /* ... */
5   // main, with objects
6   Point
7     oneone(1,1), fivetwo(5,2);
8   float dx = oneone.dx(fivetwo);
9   /* ... */
10  // main, with pointers
11  auto
12    oneonep  = make_shared<Point>(1,1),
13    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 36

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 37

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

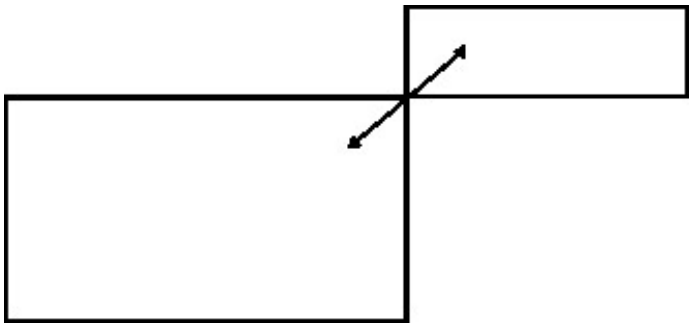
Code:

```
1 // pointer/dynrectangle.cpp
2 cout << "Area: " <<  lielow.area()
       << '\n';
3 /* ... */
4 cout << "Area: " <<  lielow.area()
       << '\n';
```

Output:

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

156. For the next exercise



Exercise 38

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

157. Memory leaks

C has a 'memory leak' problem

```
// the variable `array' doesn't exist
{
  // 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'.
(even worse if you do this in a loop!)
Java/Python have 'garbage collection': runtime impact
C++ has the best solution: smart pointers with reference counting.

158. 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"; };
};
```

159. Show constructor / destructor in actio

Code:

```
1 // pointer/ptr0.cpp
2 cout << "Outside\n";
3 {
4   thing x;
5   cout << "create done\n";
6 }
7 cout << "back outside\n";
```

Output:

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

160. Illustration 1: pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
1 // pointer/ptr1.cpp
2 cout << "set pointer1"
3      << '\n';
4 auto thing_ptr1 =
5   make_shared<thing>();
6 cout << "overwrite pointer"
7      << '\n';
8 thing_ptr1 = nullptr;
```

Output:

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

161. Illustration 2: pointer copy

Code:

```
1 // pointer/ptr2.cpp
2 cout << "set pointer2" << '\n';
3 auto thing_ptr2 =
4   make_shared<thing>();
5 cout << "set pointer3 by copy"
6      << '\n';
7 auto thing_ptr3 = thing_ptr2;
8 cout << "overwrite pointer2"
9      << '\n';
10 thing_ptr2 = nullptr;
11 cout << "overwrite pointer3"
12      << '\n';
13 thing_ptr3 = nullptr;
```

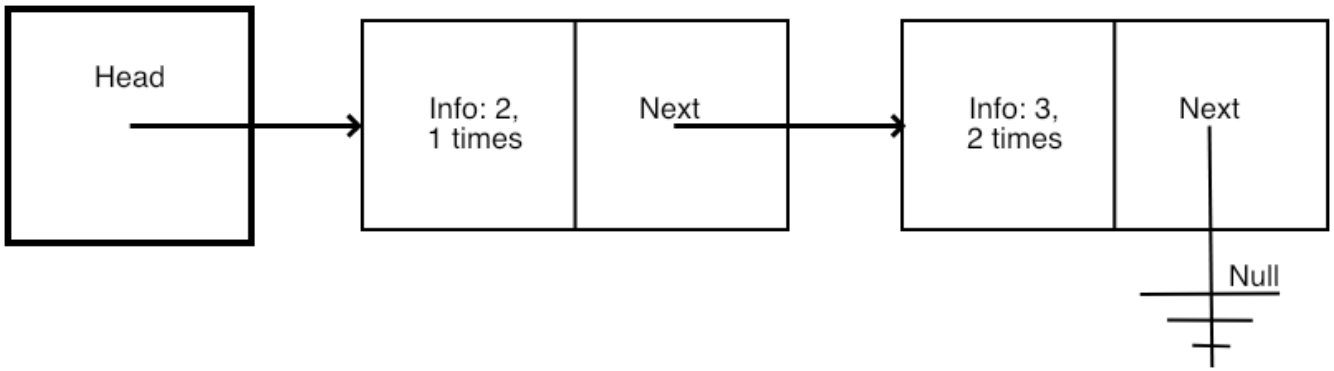
Output:

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

- 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

162. Linked list



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

163. 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.

164. Definition of Node class

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

```
1  // tree/linkshared.cpp
2  class Node {
3  private:
4    int datavalue{0},datacount{0};
5    shared_ptr<Node> next{nullptr};
6  public:
7    Node() {};
8    Node(int value,shared_ptr<Node> next=nullptr)
9      : datavalue(value),datacount(1),next(next) {};
```

A Null pointer indicates the tail of the list.

165. 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';
```

166. Recursive functions

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

167. Recursive length computation

For the list:

```
// tree/linkshared.cpp
int List::length() {
  int count = 0;
  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();
};
```

168. Iterative functions

- Recursive functions may have performance problems
- Iterative formulation possible

169. 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?)

170. 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();
  }
};
```

171. 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.

172. Definition of List class

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

```
1  // tree/linkunique.cpp
2  class List {
3  private:
4    unique_ptr<Node> head{nullptr};
5  public:
6    List() {};
```

Initially null for empty list.

173. Definition of Node class

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

```
1  // tree/linkunique.cpp
2  class Node {
3    friend class List;
4  private:
5    int datavalue{0},datacount{0};
6    unique_ptr<Node> next{nullptr};
7  public:
8    friend class List;
9    Node() {}
10   Node(int value,unique_ptr<Node> tail=nullptr)
11     : datavalue(value),datacount(1),next(move(tail)) {};
12   ~Node() { cout << "deleting node " << datavalue << '\n'; };
```

A Null pointer indicates the tail of the list.

174. Iterative computation of the list length

Use a bare pointer, which is appropriate here because it doesn't own the node. **missing snippet nodelengthiterativebare** (You will get a compiler error if you try to make `walker` a smart pointer: you can not copy a unique pointer.)

175. 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.

Lambdas

176. 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
};
```

177. Introducing: lambda expressions

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

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

New:
apply the function recipe directly:

Code:

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

Output:

```
3.8
```

178. 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.

179. Direct invocation, 1

There are uses for 'Immmediately Invoked Lambda Expression'.

Example: different constructors.

Does not work:

```
if (foo)
  MyClass x(5,5);
else
  MyClass x(6);
```

Solution:

```
auto x =
  [foo] () {
    if (foo)
      return MyClass(5,5);
    else
      return MyClass(6);
  }();
```

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

180. Direct invocation, 2

OpenMP:

```
1  const int nthreads = [] () -> int {
2    int nt;
3  #pragma omp parallel
4  #pragma omp master
5    nt = omp_get_num_threads();
6    return nt;
7  }();
```

181. Assign lambda expression to variable

Code:

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

Output:

```
3.8
8.9
```

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

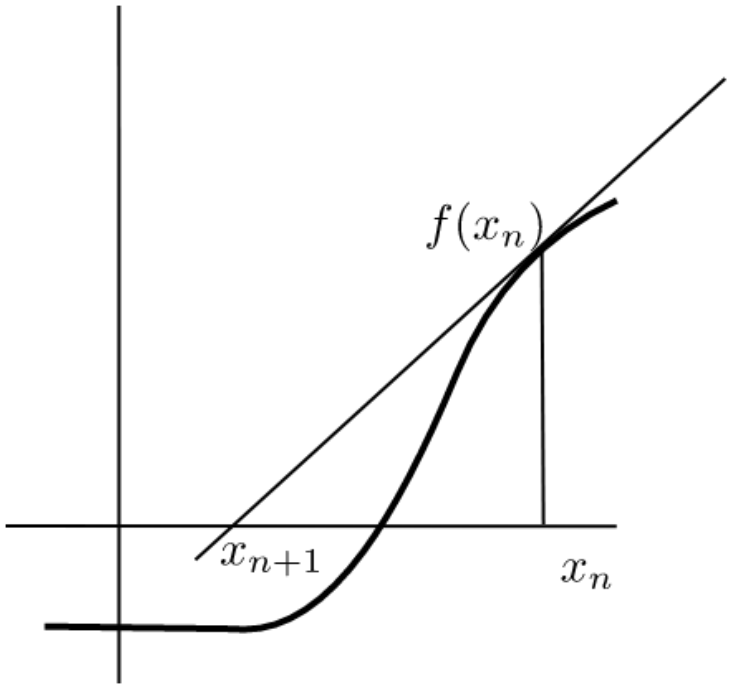
Return type could have been omitted:

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

Example of lambda usage: Newton's method

182. Newton's method

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



183. 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 39

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*

184. Function pointers

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

```
1  void f(int i) { /* something with i */ };
2  void apply_to_5( (void)(*f)(int) ) {
3      f(5);
4  }
5  int main() {
6    apply_to_5(f);
7  }
```

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

185. 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.

```
1  void apply_to_5( /* what? */ f ) {
2      f(5);
3  }
4  int main() {
5      apply_to_5
6        ( [] (double x) { cout << x; } );
7  }
```

186. Lambdas as parameter: the solution

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

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

Code:

```
1 // lambda/lambdaex.cpp
2 void apply_to_5
3     ( function< void(int) > f ) {
4   f(5);
5 }
6     /* ... */
7   apply_to_5
8     ( [] (int i) {
9       cout << "Int: " << i << '\n';
       } );
```

Output:

```
Int: 5
```

187. 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 40

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

188. Capture variable

Increment function:

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

Code:

```
1 // lambda/lambdacapture.cpp
2 int one=1;
3 auto increment_by_n =
4   [one] ( int input ) -> int {
5     return input+one;
6 };
7 cout << increment_by_n (5)  << '\n';
8 cout << increment_by_n (12)  << '\n';
9 cout << increment_by_n (25)  << '\n';
```

Output:

```
6
13
26
```

189. 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 41

- Set two variables

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

- Define a function 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?)

190. Capture value is copied

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

Code:

```
1 // lambda/lambdacapture.cpp
2 int inc;
3 cin >> inc;
4 auto increment =
5   [inc] ( int input ) -> int {
6     return input+inc;
7   };
8 cout << "increment by: " << inc << '\n';
9 cout << "1 -> "
10      << increment(1) << '\n';
11 inc = 2*inc;
12 cout << "1 -> "
13      << increment(1) << '\n';
```

Output:

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

Exercise 42

Extend the newton exercise to compute roots in a loop:

```
// newton/newton-lambda.cpp
  for (int n=2; n<=8; ++n) {
    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.

191. 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 43

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 for the gradient and pass it to the function `newton_root` you coded earlier.
This is polymorphism: you now have two definition for the same function. They differ in the number of arguments.

192. Lambda in object

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

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

193. Illustration

The above code in use:

Code:

```
1 // lambda/lambdafun.cpp
2 cout << "Give a divisor: ";
3 cin >> divisor; cout << '\n';
4 cout << ".. using " << divisor
5      << '\n';
6 auto is_divisible =
7   [divisor] (int i) -> bool {
8     return i%divisor==0; };
9 SelectedInts multiples(
       is_divisible );
10 for (int i=1; i<50; ++i)
11   multiples.add(i);
```

Output:

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

Advanced topics

194. Capture by value

Normal capture is by value:

Code:

```
1 // lambda/lambdacapture.cpp
2 int one=1;
3 auto increment_by_n =
4   [one] ( int input ) -> int {
5     return input+one;
6 };
7 cout << increment_by_n (5)  << '\n';
8 cout << increment_by_n (12)  <<
      '\n';
9 cout << increment_by_n (25)  <<
      '\n';
```

Output:

```
6
13
26
```

195. Capture by reference

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

```
int count=0;
auto count_if_f =
    [&count] (int i) {
      if (f(i)) count++; }
for ( int i : int_data )
  count_if_f(i);
cout << "We counted: " << count;
```

(See the algorithm header, section **??**.)

196. Lambdas vs function pointers

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

Code:

```
1 // lambda/lambdacptr.cpp
2 int cfun_add1( int i ) {
3   return i+1; };
4 int apply_to_5( int(*f)(int) ) {
5   return f(5); };
6 //codesnippet  end
7     /* ... */
8   auto lambda_add1 =
9     [] (int i) { return i+1; };
10  cout << "C ptr: "
11       << apply_to_5(&cfun_add1)
12       << '\n';
13  cout << "Lambda: "
14       << apply_to_5(lambda_add1)
15       << '\n';
```

Output:

```
missing snippet
../../code/func/lambdacptr.runout
```

197. Use in algorithms

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

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

See later.

Union-like things

Tuples

198. Example for this lecture

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

199. Returning two things

Simple solution:

```
// union/optroot.cpp
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = std::sqrt(x);
  return true;
};
    /* ... */
  for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
      cout << "Root is " << x << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

Other solution: tuples

200. Function returning tuple

How do you return two things of different types?

```
1  #include <tuple>
2  using std::make_tuple, std::tuple;
3
4  tuple<bool,float> maybe_root1(float x) {
5    if (x<0)
6      return make_tuple<bool,float>(false,-1);
7    else
8      return make_tuple<bool,float>(true,sqrt(x));
9  };
10
```

(not the best solution for the 'root' code)

201. Returning tuple with type deduction

Return type deduction:

```
1  // stl/tuple.cpp
2  auto maybe_root1(float x) {
3    if (x<0)
4      return make_tuple
5        <bool,float>(false,-1);
6    else
7      return make_tuple
8        <bool,float>
9          (true,sqrt(x));
10 };
```

Alternative:

```
1  // stl/tuple.cpp
2  tuple<bool,float>
3      maybe_root2(float x) {
4    if (x<0)
5      return {false,-1};
6    else
7      return {true,sqrt(x)};
8  };
```

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

202. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
1 // stl/tuple.cpp
2 auto [succeed,y] = maybe_root2(x);
3 if (succeed)
4   cout << "Root of " << x
5        << " is " << y << '\n';
6 else
7   cout << "Sorry, " << x
8        << " is negative" << '\n';
```

Output:

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

This is known as structured binding.

203. C++11 style tuples

```
#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

204. 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);
};
    /* ... */
  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';
```

205. Create optional

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

optional<float> f {
  if (something)
  // result if success
  return 3.14;
  else
  // indicate failure
  return {};
}
```

Expected (C++23)

206. Expected

Expect double, return info string if not:

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

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

Variants

207. 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.

208. Variant methods

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

Get the index of what the variant contains:

```
1 // union/intdoublestring.cpp
2 union_ids = 3.5;
3 switch ( union_ids.index() ) {
4 case 1 :
5   cout << "Double case: " << std::get<double>(union_ids) << '\n';
6 }
```

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

(Takes pointer to variant, returns pointer to value)

Exercise 44

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.

Code:

```
1 // union/quadratic.cpp
2   for ( auto coefficients :
3           { quadratic{.a=2.0,
      .b=1.5, .c=2.5},
4             quadratic{.a=1.0,
      .b=4.0, .c=4.0},
5             quadratic{.a=2.2,
      .b=5.1, .c=2.5}
6           } ) {
7     auto result =
       compute_roots(coefficients);
```

Output:

```
With a=2 b=1.5 c=2.5
No root
With a=2.2 b=5.1 c=2.5
Root1: -0.703978 root2:
     -1.6142
With a=1 b=4 c=4
Single root: -2
```

Iterators

209. 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 ):
```

210. Loop algorithms

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

Ranges: iteratable things such as vectors

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

C++20 ranges

211. Range over vector

With

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
1  // rangestd/range.cpp
2  #include <algorithm>
3  #include <ranges>
4      /* ... */
5    std::ranges::for_each
6      ( v,
7        [] (int i) {
8          cout << i << " ";
9        }
10       );
```

Output:

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

212. Ranged algorithm

With

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
1 // rangestd/range.cpp
2 count = 0;
3 std::ranges::for_each
4   ( v,
5     [&count] (int i) {
6       count += (i<5); }
7     );
8 cout << "Under five: "
9      << count << '\n';
```

Output:

```
Under five: 3
```

213. Range composition

Pipeline of ranges and views:

```
// rangestd/range.cpp
vector<int> generate_data() { return {2,3,4,5,6,7}; };
    /* ... */
  auto v = generate_data();
```

Code:

```
1 // rangestd/range.cpp
2 count = 0;
3 std::ranges::for_each
4   ( v
5     | std::ranges::views::drop(1),
6     [&count] (int i) {
7       count += (i<5); }
8     );
9 cout << "minus first: "
10      << count << '\n';
```

Output:

```
minus first: 2
```

'pipe operator'

214. Iota and take

Code:

```
1 // rangestd/iota.cpp
2 #include <ranges>
3 namespace rng = std::ranges;
4     /* ... */
5   for ( auto n :
6           rng::views::iota(2,6) )
7     cout << n << '\n';
8   cout << "===\n";
9   for ( auto n :
10          rng::views::iota(2)
11          | rng::views::take(4) )
12    cout << n << '\n';
```

Output:

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

Exercise 45: Iota and take

Rewrite the second loop of the previous slide using an algorithm, and no explicit loop.

215. Filter

Take a range, and make a new one of only the elements satisfying some condition:

Code:

```
1 // rangestd/filter.cpp
2 vector<float> numbers
3   {1,-2.2,3.3,-5,7.7,-10};
4 auto pos_view =
5   numbers
6   | std::ranges::views::filter
7     ( [] (int i) -> bool {
8         return i>0; }
9     );
10 for ( auto n : pos_view )
11   cout << n << " ";
12 cout << '\n';
```

Output:

```
1 3.3 7.7
```

Exercise 46: Element counting

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

216. Range composition

Code:

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

Output:

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

217. Quantor-like algorithms

Code:

```
1 // rangestd/of.cpp
2 vector<int> integers{1,2,3,5,7,10};
3 auto any_even =
4   std::ranges::any_of
5     ( integers,
6       [=] (int i) -> bool {
7         return i%2==0; }
8     );
9 if (any_even)
10   cout << "there was an even\n";
11 else
12   cout << "none were even\n";
```

Output:

```
there was an even
```

Also *all_of*, *none_of*

218. Reductions

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

Exercise 47: Perfect numbers

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.

Iterators

219. Iterate without iterators

```
vector data{2,3,1};
sort( begin(data),end(data) ); // open to accidents
ranges::sort(data);
```

220. Begin and end iterator

Use independent of looping:

Code:

```
1  // stl/iter.cpp
2      vector<int> v{1,3,5,7};
3      auto pointer = v.begin();
4      cout << "we start at "
5           << *pointer << '\n';
6      ++pointer;
7      cout << "after increment: "
8           << *pointer << '\n';
9
10     pointer = v.end();
11     cout << "end is not a valid
        element: "
12          << *pointer << '\n';
13     pointer--;
14     cout << "last element: "
15          << *pointer << '\n';
```

Output:

```
we start at 1
after increment: 3
end is not a valid
     element: 0
last element: 7
```

221. Erase at/between iterators

Erase from start to before-end:

Code:

```
1 // iter/iter.cpp
2 vector<int> counts{1,2,3,4,5,6};
3 vector<int>::iterator second =
      counts.begin()+1;
4 auto fourth = second+2;
5 counts.erase(second,fourth);
6 cout << counts[0]
7      << "," << counts[1] << '\n';
```

Output:

```
1,4
```

(Also erasing a single element without end iterator.)

222. Insert at iterator

Insert at iterator: value, single iterator, or range:

Code:

```
1 // iter/iter.cpp
2 vector<int> counts{1,2,3,4,5,6},
3   zeros{0,0};
4 auto after_one = zeros.begin()+1;
5 zeros.insert
6   ( after_one,
7     counts.begin()+1,
8     counts.begin()+3 );
9 cout << zeros[0] << ","
10      << zeros[1] << ","
11      << zeros[2] << ","
12      << zeros[3]
13      << '\n';
```

Output:

```
0,2,3,0
```

223. Iterator arithmetic

```
auto first = myarray.begin();
first += 2;
auto last  = myarray.end();
last  -= 2;
myarray.erase(first,last);
```

Algorithms with iterators

224. Reduction operation

Default is sum reduction:

Code:

```
1 // stl/reduce.cpp
2 #include <numeric>
3 using std::accumulate;
4     /* ... */
5     vector<int> v{1,3,5,7};
6     auto first = v.begin();
7     auto last  = v.end();
8     auto sum =
       accumulate(first,last,0);
9     cout << "sum: " << sum << '\n';
```

Output:

```
sum: 16
```

225. Reduction with supplied operator

Supply multiply operator:

Code:

```
1  // stl/reduce.cpp
2  using std::multiplies;
3      /* ... */
4      vector<int> v{1,3,5,7};
5      auto first = v.begin();
6      auto last  = v.end();
7      ++first; last--;
8      auto product =
9        accumulate(first,last,2,
10                  multiplies<>());
11     cout << "product: " << product
        << '\n';
```

Output:

```
product: 30
```

226. Custom reduction function

```
// stl/reduce.cpp
class x {
public:
  int i,j;
  x() {};
  x(int i,int j) : i(i),j(j)
    {};
};
```

```
// stl/reduce.cpp
    std::vector< x > xs(5);
    auto xxx =
      std::accumulate
      ( xs.begin(),xs.end(),0,
        [] ( int init,x x1 )
    -> int { return x1.i+init;
    }
        );
```

Write your own iterator

227. Vector iterator

Range-based iteration

```
for ( auto element : vec ) {
  cout << element;
}
```

is syntactic sugar around iterator use:

```
for (std::vector<int>::iterator elt_itr=vec.begin();
      elt_itr!=vec.end(); ++elt_itr) {
  element = *elt_itr;
  cout << element;
}
```

228. Custom iterators, 0

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
  ... e ...
```

for:

```
for ( vector<float>::iterator
    e=v.begin();
      e!=v.end(); e++ )
  ... *e ...
```

If we want

```
for ( auto e : my_object )
    ... e ...
```

we need an iterator class with methods such as *begin*, *end*, * and ++.

229. Custom iterators, 1

Ranging over a class with iterator subclass

Class:

```
// loop/iterclass.cpp
class NewVector {
protected:
  // vector data
  int *storage;
  int s;
    /* ... */
public:
  // iterator stuff
  class iter;
  iter begin();
  iter end();

};
```

Main:

```
// loop/iterclass.cpp
NewVector v(s);
  /* ... */
for ( auto e : v )
  cout << e << " ";
```

230. Custom iterators, 2

Random-access iterator:

```
// loop/iterclass.cpp
NewVector::iter& operator++();
int& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-( const NewVector::iter& other ) const;
NewVector::iter& operator+=( int add );
```

Exercise 48

Write the missing iterator methods. Here's something to get you started.

```
// loop/iterclass.cpp
class NewVector::iter {
private: int *searcher;
    /* ... */
NewVector::iter::iter( int *searcher )
  : searcher(searcher) {};
NewVector::iter NewVector::begin() {
  return NewVector::iter(storage); };
NewVector::iter NewVector::end()   {
  return NewVector::iter(storage+NewVector::s); };
```

Namespaces

231. 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

232. 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.
```

233. 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;
}
```

234. Defining a namespace

Introduce new namespace:

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

235. 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();
```

236. 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;
}
```

(Why?)

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;
}
```

237. Guideline

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

Example

238. 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"
using namespace geo;
int main() {
  // std vector of geom segments:
  std::vector< segment > segments;
  segments.push_back( segment( point(1,1),point(4,5) ) );
```

What would the implementation of this be?

239. Namespace'd declarations

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

240. Namespace'd implementations

```
// namespace/geolib.cpp
namespace geo {
  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

241. 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>
```

242. Example: function

Definition:

```
// template/func.cpp
template <typename T>
void function( T x ) {
  cout << std::sqrt(x)-1.772 << '\n';
};
```

We use this with a templated function:

Code:

```
1 // template/func.cpp
2 function<float>( 3.14f );
3 function<double>( 3.14 );
```

Output:

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

Exercise 49

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 *epsilon* so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
1 // template/eps.cpp
2 float float_eps;
3 epsilon(float_eps);
4 cout << "Epsilon float: "
5      << setw(10) << setprecision(4)
6      << float_eps << '\n';
7
8 double double_eps;
9 epsilon(double_eps);
10 cout << "Epsilon double: "
11      << setw(10) << setprecision(4)
12      << double_eps << '\n';
```

Output:

```
Epsilon float:
     1.0000e-07
Epsilon double:
     1.0000e-15
```

Hint: you may need to cast scalars to the appropriate type

243. 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

244. Throw an integer

Throw an integer error code:

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

245. Catching an exception

Catch an integer:

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

Exercise 50

Revisit the prime generator class (exercise 73) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 2.)

Code:

```
1 // primes/genx.cpp
2 try {
3   do {
4     auto cur = primes.nextprime();
5     cout << cur << '\n';
6   } while (true);
7 } catch ( string s ) {
8   cout << s << '\n';
9 }
```

Output:

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

246. 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
}
```

247. Catch any exception

Catch exceptions without specifying the type:

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

248. 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!

249. Exceptions in constructors

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

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

250. More about exceptions

- Functions can define what exceptions they throw:

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

- 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 66.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the exception header file.
- Keyword `noexcept`:

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

- There is no exception thrown when dereferencing a `nullptr`.

251. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
1 // object/exceptdestruct.cpp
2 class SomeObject {
3 public:
4   SomeObject() {
5     cout << "calling the
       constructor"
6          << '\n'; };
7   ~SomeObject() {
8     cout <<  "calling the
       destructor"
9          << '\n'; };
10 };
11    /* ... */
12  try {
13    SomeObject obj;
14    cout << "Inside the nested
       scope" << '\n';
15    throw(1);
16  } catch (...) {
17    cout << "Exception caught" <<
       '\n';
```

Output:

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

252. 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;
```

253. 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. );
```

254. 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

255. 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();
```

256. Type deduction in functions

Return type of functions can be deduced in C++17:

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

257. Type deduction in methods

Return type of methods can be deduced in C++17:

```
1  // auto/plainget.cpp
2  class A {
3  private: float data;
4  public:
5    A(float i) : data(i) {};
6    auto &access() {
7      return data; };
8    void print() {
9      cout << "data: " << data << '\n'; };
10 };
```

258. Auto and references, 1

Demostrating that `auto` discards references from the rhs:

Code:

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

Output:

```
data: 5.7
```

259. Auto and references, 2

Combine `auto` and references:

Code:

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

Output:

```
data: 6.7
```

260. Auto and references, 3

For good measure:

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

Casts

261. 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

262. Static cast

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

Code:

```
1 // cast/intlong.cpp
2 long int hundredg = 100000000000;
3 cout << "long number:      "
4      << hundredg << '\n';
5 int overflow;
6 overflow =
       static_cast<int>(hundredg);
7 cout << "assigned to int: "
8      << overflow << '\n';
```

Output:

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

263. 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'; };
};
```

264. Cast to derived class

This is how to do it:

Code:

```
1 // cast/toderived.cpp
2 cout << "Dynamic cast to Derived\n";
3 Derived object1;
4 f(object1);
5 Erived object2;
6 f(object2);
```

Output:

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

265. Cast to derived class, the wrong way

Do not use this function g:

Code:

```
1 // cast/toderivedp.cpp
2 Base *object = new Derived();
3 g(object);
4 Base *nobject = new Erived();
5 g(nobject);
```

Output:

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

Const

266. 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).

267. 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;
}
```

268. Const ref parameters

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

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

269. No side-effects

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

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

270. Const polymorphism

Const and non-const version of `at`:

Code:

```
1 // const/constat.cpp
2 class has_array {
3 private:
4   vector<float> values;;
5 public:
6   has_array(int l,float v)
7     : values(vector<float>(l,v)) {};
8   auto& at(int i) {
9     cout << "var at" << '\n';
10    return values.at(i); };
11  const auto& at (int i) const {
12    cout << "const at" << '\n';
13    return values.at(i); };
14  auto sum() const {
15    float p;
16    for ( int i=0; i<values.size();
       ++i)
17      p += at(i);
18    return p;
19  }
20 };
```

Output:

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

Exercise 51

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?

271. 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;
}
```

272. 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.)

273. Mutable example

Code:

```
1 // object/mutable.cpp
2 class has_stuff {
3 private:
4   mutable optional<complicated>
        thing = {};
5 public:
6   const complicated& get_thing()
        const {
7     if ( not thing.has_value() )
8       thing = complicated(5);
9     else cout << "thing already
        there\n";
10    return thing.value();
11  };
12 };
```

Output:

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

More STL

Complex

Complex numbers

274. 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 52

Rewrite your Newton program so that it works for complex numbers:

```
// 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`; see section **??**.

Templated functions

275. Templatized Newton, first attempt

You can templatize your Newton function and derivative:

```
// 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

```
// 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 53

Update your Newton program with templates. If you have it working for `double`, try using *complex*<`double`>. Does it work?

Exercise 54

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

How about $\sqrt{-2}$?

Exercise 55

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>>
  ( [] (complex<double> x) -> complex<double> {
       return x*x + static_cast<complex<double>>(2); },
    [] (complex<double> x) -> complex<double> {
       return x * static_cast<complex<double>>(2); },
    complex<double>{.1,.1}
    )
     << '\n';
```

Random

276. 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.

277. Random workflow

Use a new header

```
#include <random>
```

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.

278. 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);
```

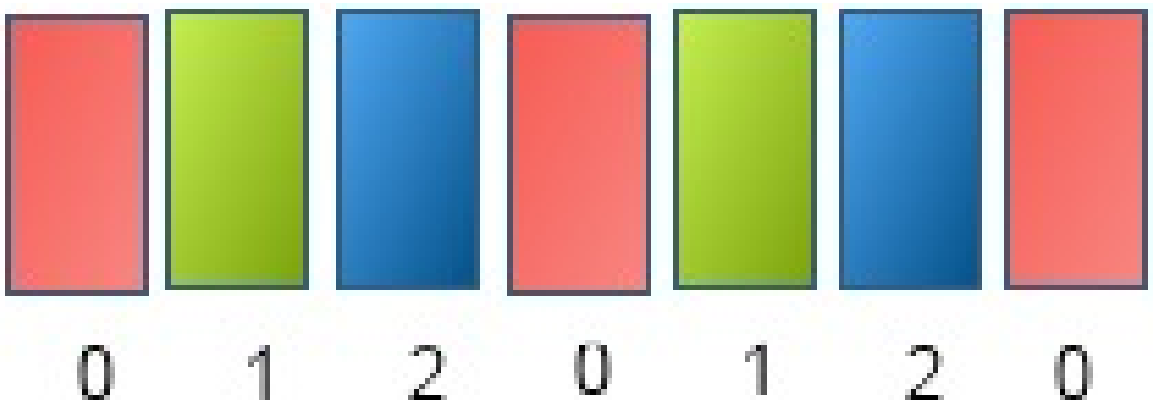
- Do not use the old C-style `random`!

279. 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



280. 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
```

281. 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);
```

282. Local engine

Wrong approach: random generator local in the function.

Code:

```
1 // rand/static.cpp
2 int nonrandom_int(int max) {
3   std::default_random_engine engine;
4   std::uniform_int_distribution<>
      ints(1,max);
5   return ints(engine);
6 };
7     /* ... */
8   // call `nonrandom_int three times
```

Output:

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

Generator gets recreated in every function call.

Exercise 56

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);
};
```

283. Global engine

Good approach: random generator static in the function.

Code:

```
1 // rand/static.cpp
2 int realrandom_int(int max) {
3   static
       std::default_random_engine
       static_engine;
4   std::uniform_int_distribution<>
       ints(1,max);
5   return ints(static_engine);
6 };
```

Output:

```
Three ints: 15, 98, 70.
```

A single instance is ever created.

284. What does 'static' do?

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

285. Class with static member

Class that counts how many objects have been generated:

Code:

```
1 // object/static.cpp
2 class Thing {
3 private:
4   static inline int nthings{0};
5   int mynumber;
6 public:
7   Thing() {
8     mynumber = nthings++;
9     cout << "I am thing "
10         << mynumber << '\n';
11  };
12 };
```

Output:

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

Optional exercise 57

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.

286. Generator in a class

Note the use of `static`:

```
// 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:

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

Limits

287. Templated functions for limits

Use header file `limits`:

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

cout << numeric_limits<long>::max();
```

288. Some limit values

Code:

```
1 // stl/limits.cpp
2 cout << "Signed int: "
3      << numeric_limits<int>::min()
       << " "
4      << numeric_limits<int>::max()
5      << '\n';
6 cout << "Unsigned    "
7      << numeric_limits<unsigned
       int>::min() << " "
8      << numeric_limits<unsigned
       int>::max()
9      << '\n';
10 cout << "Single      "
11     <<
       numeric_limits<float>::denorm_min()
       << " "
12     <<
       numeric_limits<float>::min()
       << " "
13     << numeric_limits<float>::max()
14     << '\n';
15 cout << "Double      "
16     <<
```

Output:

```
Signed int: -2147483648
     2147483647
Unsigned    0 4294967295
Single      1.4013e-45
     1.17549e-38
     3.40282e+38
Double
     4.94066e-324
     2.22507e-308
     1.79769e+308
```

289. Limits of floating point values

- The largest number is given by `max`; use `lowest` for 'most negative'.
- The smallest denormal number is given by `denorm_min`.
- `min` is the smallest positive number that is not a denormal;
- There is an `epsilon` function for machine precision:

Code:

```
1 // stl/limits.cpp
2 cout << "Single lowest "
3      <<
       numeric_limits<float>::lowest()
4      << " and epsilon "
5      <<
       numeric_limits<float>::epsilon()
6      << '\n';
7 cout << "Double lowest "
8      <<
       numeric_limits<double>::lowest()
9      << " and epsilon "
10     <<
       numeric_limits<double>::epsilon()
11     << '\n';
```

Output:

```
Single lowest
     -3.40282e+38 and
     epsilon 1.19209e-07
Double lowest
     -1.79769e+308 and
     epsilon 2.22045e-16
```

Random numbers

290. Random floats

Random numbers from the unit interval:

```
// rand/xrand.cpp
// seed the generator
std::random_device r;
// set the default random number generator
std::default_random_engine generator{r()};
// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

for ( int i=0; i<5; i++)
  cout << "random: "
       << distribution(generator)
       << '\n';
```

291. 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

292. 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"
```

293. Date

coming in C++20

File system

294. file system

```
#include <filesystem>
```

including directory walker

Regular expressions

295. Example

Code:

```
1 // regexp/regexp.cpp
2 auto cap = regex("[A-Z][a-z]+");
3 for ( auto n :
4       {"Victor", "aDam", "DoD"}
5       ) {
6   auto match =
7     regex_match( n, cap );
8   cout << n;
9   if (match) cout << ": yes";
10  else       cout << ": no" ;
11  cout << '\n';
12 }
```

Output:

```
Looks like a name:
Victor: yes
aDam: no
DoD: no
```

C++20 modules

296. Modules

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

Unit testing

Intro to testing

297. 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 ...

298. 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.

299. 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

300. 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

301. 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.

302. 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

303. 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.

304. 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 58

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

305. Boolean tests

Test a boolean expression:

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

306. Output for failing tests

Run the tester:

Code:

```
1 // catch/false.cpp
2 #define CATCH_CONFIG_MAIN
3 #include "catch2/catch_all.hpp"
4
5 int five() { return 6; }
6
7 TEST_CASE( "needs to be 5" ) {
8     REQUIRE( five()==5 );
9 }
```

Output:

```
Randomness seeded to:
     235485692

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
false is a Catch2
     v3.1.1 host
     application.
Run with -? for options

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

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

======================================
```

307. 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 );
    REQUIRE( f(n)>0 );
}
```

308. 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

309. 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) );
}
```

310. 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 );
```

311. Tests with code in common

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

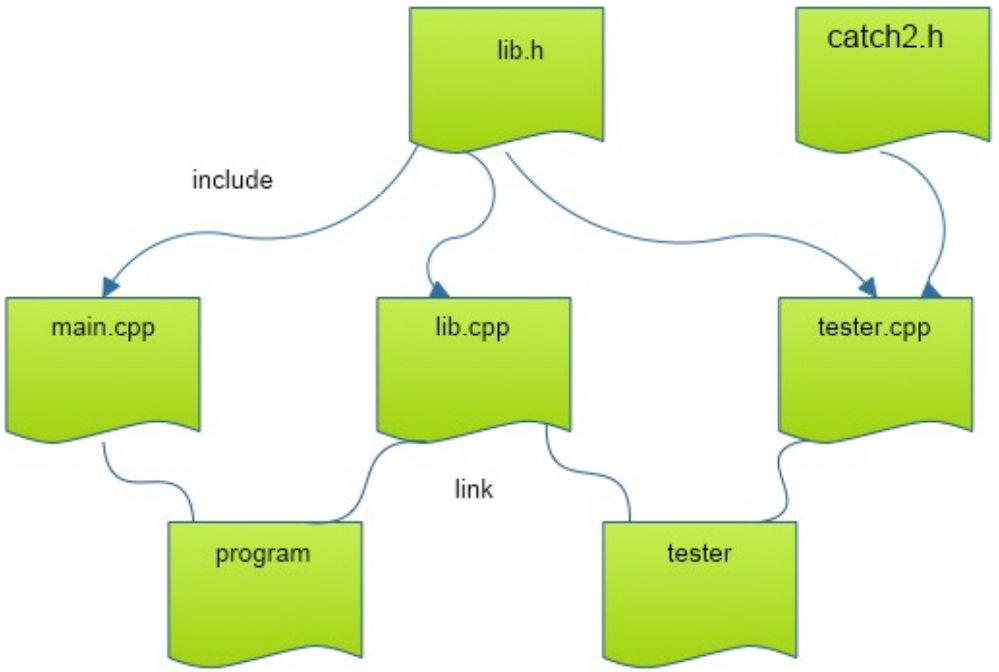
```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

Catch2 file structure

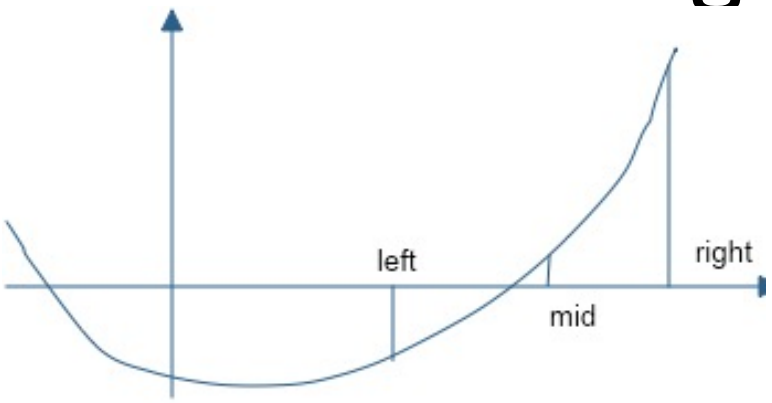
312. Realistic setup

- 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

313. 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.

314. 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 59: 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.

315. 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 60: 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) );
```

316. 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 61: 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) );
```

317. 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 62: 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?

318. 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 63: 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 ) ) );
```

319. 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:
  $|f(y)| <$ `prec`.

Exercise 64: Put it all together

Make this call work:

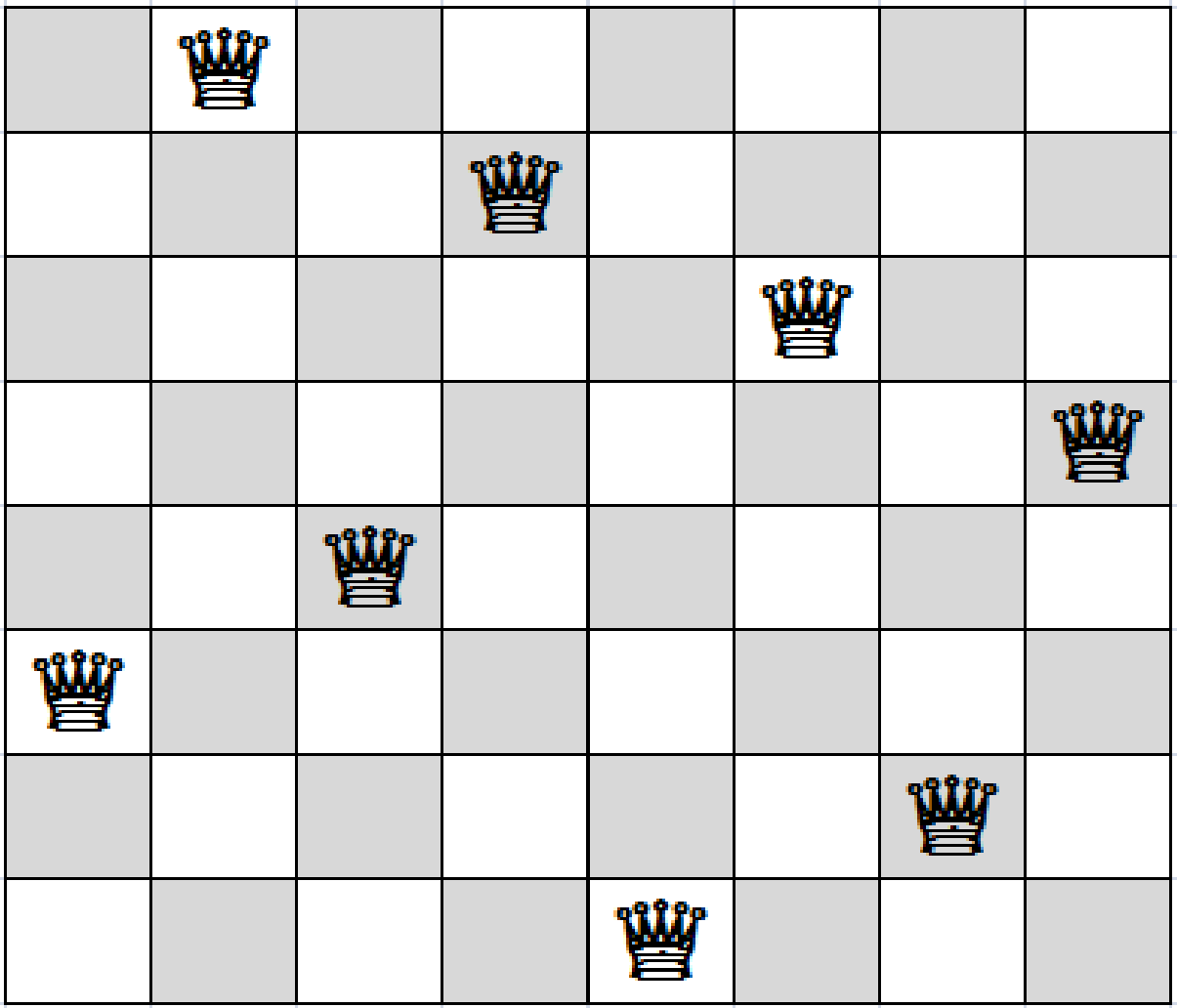
```
// bisect/zeroclassmain.cpp
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

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

TDD example: Eight queens

320. Classic problem

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



321. 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.

322. 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 65: 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 66: 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 67: 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 68: 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 69: 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 70: 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 71: 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 72: 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 73: 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() );
}
```

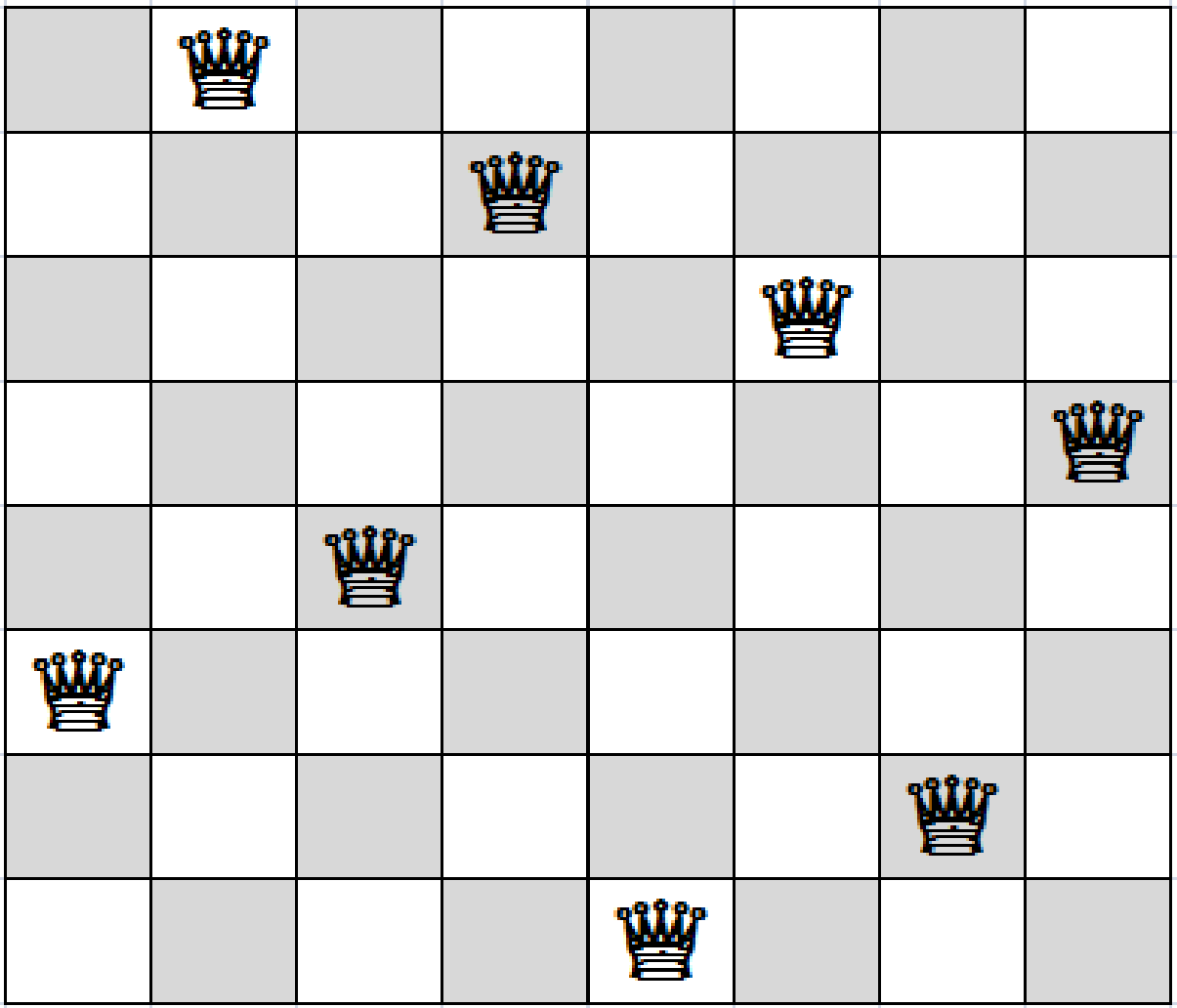
Exercise 74: O

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

Eight queens problem by TDD (using objects)

323. Problem statement

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

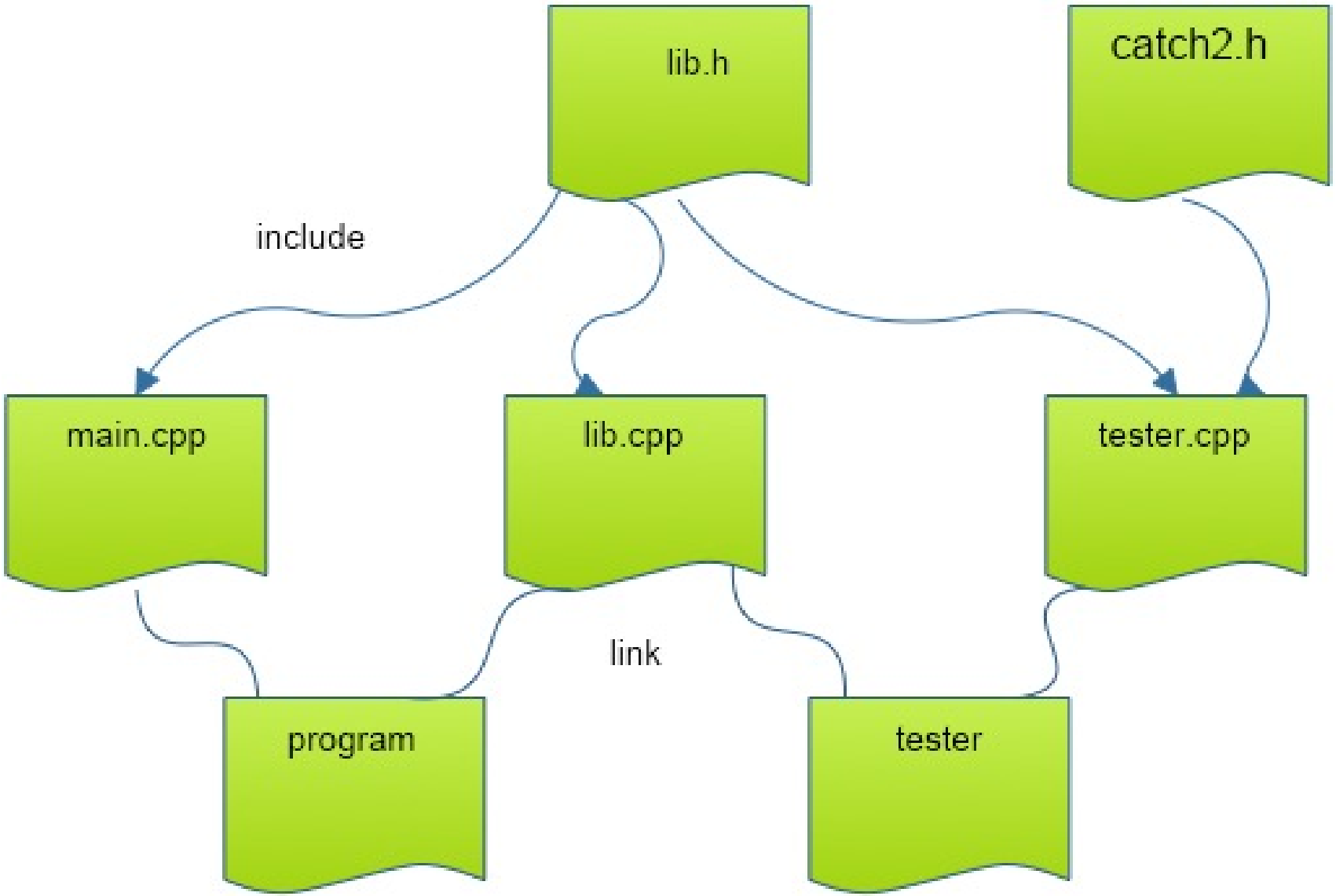


324. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

325. File structure



326. 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 75: Board object

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

Exercise 76: 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 77: 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);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Exercise 78: 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 79: 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 80: 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

327. 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.

328. C++11

- `auto`
- Range-based for.
- Lambdas.
- Variadic templates.
- Smart pointers.
- `constexpr`

329. 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.

330. C++17

- Optional; section **??**.
- Structured binding declarations as an easier way of dissecting tuples; section 275.
- Init statement in conditionals; section **??**.

331. 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 **??**.

332. C++23

- *md_span*