C++ for C Programmers

Victor Eijkhout

TACC HPC Training 2023

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://tinyurl.com/vle322course`

4. General note about syntax

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

```
icpc    -std=c++17 yourprogram.cxx
g++     -std=c++17 yourprogram.cxx
clang++ -std=c++17 yourprogram.cxx
```

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

```
alias icpc='icpc -std=c++17'
et cetera
```

5. Build with Cmake

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

add_executable( ${PROGRAM_NAME} ${PROGRAM_NAME}.cxx )
target_compile_features( ${PROGRAM_NAME}
    PRIVATE cxx_std_17 )
```

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: 'post-modern' C++.
  Ratified, but only partly implemented.
- C++23/26: 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:

```
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. Main

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";
}
```

Later: `fmtlib`

12. 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

13. Big and small changes

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

Parameter passing

14. 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’

15. 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'.

16. 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)

17. Reference

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

Code:

```
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << '\n';
i *= 2;
cout << i << "," << ri << '\n';
ri -= 3;
cout << i << "," << ri << '\n';
```

Output:

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

(You will not use references often this way.)

18. Create reference by initialize

Correct:

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

Not correct:

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

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

19. 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.

20. Parameter passing by reference

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

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

Reference syntax is cleaner than C 'pass by reference'

21. Pass by reference example 1

Code:

```
void f( int &i ) {
  i = 5;
}
int main() {

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

Output:

```
5
```

Compare the difference with leaving out the reference.

22. 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' ....
```

23. 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:

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

Output:

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

Optional exercise 2

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

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

Code:

```
cout << number;
if
     (is_divisible(number,divisor,remain
  cout << " is divisible by ";
else
  cout << " has remainder "
       << remainder << " from ";
cout << divisor << '\n';
```

Output:

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

More about functions

24. 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.

25. 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
```

26. 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

27. 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.

28. Running example

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

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

Exercise 3

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

29. Object functionality

Small illustration: point objects.

Code:

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

Output:

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

Note the ‘dot’ notation.

Exercise 4

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

30. 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( /* ... */ );
```

31. Construct an object

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

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

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

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

32. 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

33. Class methods

Let's define method *distance*.

Definition in the class:

```
class Point {
  /* stuff */
  double distance_to_origin() {
    return sqrt(x*x + y*y); };
}
```

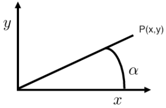
Use in the program:

```
Point pt(5,12);
double
  s = pt.distance_to_origin();
```

- 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.cxx` *in the repository*

Exercise 6

Make a class `GridPoint` which can 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.

34. Food for thought: constructor vs data

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

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

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

Note: no change to outward API.

Exercise 7

Discuss the pros and cons of this design:

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

35. Data access in methods

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

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

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

Exercise 8

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

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

computes the distance between them.

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"`

36. Methods that alter the object

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

Code:

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

Output:

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

Data initialization

37. 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.

38. Data initialization

The naive way:

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

The preferred way:

```
class Point {
private:
  double x,y;
public:
  Point( double userx,
         double usery )
    : x(userx),y(usery) {
  }
```

Interaction between objects

39. Methods that create a new object

Code:

```
class Point {
  /* ... */
  Point scale( double a ) {
    auto scaledpoint =
        Point( x*a, y*a );
    return scaledpoint;
  };
  /* ... */
  cout << "p1 to origin "
       << p1.dist_to_origin()
       << '\n';
  Point p2 = p1.scale(2.);
  cout << "p2 to origin "
       << p2.dist_to_origin()
       << '\n';
```

Output:

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

Note the 'anonymous object' in the assignment

40. Anonymous objects

Create a point by scaling another point:

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

Two ways of handling the `return` statement:

Naive:

```
Point Point::scale( double a )
    {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

Creates point, copies it to *new_point*

Better:

```
Point Point::scale( double a )
    {
  return Point( x*a, y*a );
};
```

Creates point, moves it directly to *new_point*

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

Optional 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?

41. 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:

```
class IamOne {
private:
  int i=1;
public:
  void print() {
    cout << i << '\n';
  };
};
  /* ... */
  IamOne one;
  one.print();
```

Output:

```
1
```

42. Default constructor

Refer to point definition: 43
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 (g++; different for intel):

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

43. Default constructor

The problem is with *p2*:

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

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

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

- as soon as you define a constructor, the default constructor goes away;
- you need to redefine the default constructor:

  ```
  Point() {};
  Point( double x,double y )
    : x(x),y(y) {};
  ```

  (but only if you really need it.)

44. Other way

```
Point() = default;
Point( double x,double y )
  : x(x),y(y) {};
```

Also:

```
Point() = delete;
```

to disable.

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;
```

45. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

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

Output:

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

46. 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`

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:

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

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. Use the `primegenerator` class you developed in exercise 69.

This is a great exercise for a top-down approach!

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.

47. A Goldbach corollary

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

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

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

In particular this holds for each prime number:

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

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. 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.

Advanced stuff

48. Direct alteration of internals

Return a reference to a private member:

```
class Point {
private:
  double x,y;
public:
  double &x_component() { return x; };
};
int main() {
  Point v;
  v.x_component() = 3.1;
}
```

Only define this if you need to be able to alter the internal entity.

49. Reference to internals

Returning a reference saves you on copying.
Prevent unwanted changes by using a ‘const reference’.

```
class Grid {
private:
  vector<Point> thepoints;
public:
  const vector<Point> &points() const {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

50. Access gone wrong

We make a class for points on the unit circle

```
class UnitCirclePoint {
private:
  float x,y;
public:
  UnitCirclePoint(float x) {
    setx(x); };
  void setx(float newx) {
    x = newx; y = sqrt(1-x*x);
  };
```

You don't want to be able to change just one of x,y!

In general: enforce invariants on the members.

51. Const functions

A function can be marked as const:
it does not alter class data,
only changes are through return and parameters

52. 'this' pointer to the current object

Inside an object, a pointer to the object is available as `this`:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint; // `this' redundant!
  };
};
```

53. 'this' use

You don't often need the `this` pointer. Example: you need to call a function inside a method that needs the object as argument)

```
/* forward definition: */ class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

(Rare use of dereference star)

Operator overloading

54. Operator overloading

Syntax:

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

For instance:

Code:

```
Point Point::operator*(double f) {
    return Point(f*x,f*y);
};
  /* ... */
  cout << "p1 to origin "
       << p1.dist_to_origin() <<
    '\n';
  Point scale2r = p1*2.;
  cout << "scaled right: "
       << scale2r.dist_to_origin()
    << '\n';
  // ILLEGAL Point scale2l = 2.*p1;
```

Output:

```
p1 to origin 2.23607
scaled right: 4.47214
```

Can also:

```
void Point::operator*=(double factor);
```

Exercise 14

Revisit exercise 9 and replace the `add` and `scale` functions by overloaded operators.

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

55. Constructors and contained classes

Finally, if a class contains objects of another class,

```
class Inner {
public:
  Inner(int i) { /* ... */ }
};
class Outer {
private:
  Inner contained;
public:
};
```

56. When are contained objects created?

```
Outer( int n ) {
  contained = Inner(n);
};
```

1. This first calls the default constructor
2. then calls the `Inner(n)` constructor,
3. then copies the result over the `contained` member.

```
Outer( int n )
  : contained(Inner(n)) {
    /* ... */
};
```

1. This creates the `Inner(n)` object,
2. placed it in the `contained` member,
3. does the rest of the constructor, if any.

57. Copy constructor

- Default defined copy and 'copy assignment' constructors:

```
some_object x(data);
some_object y = x;
some_object z(x);
```

- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed.

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v
         << '\n';
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v
         << '\n';
    mine = v; };
  void printme() {
    cout << "I have: " << mine
         << '\n'; };
};
```

58. Copy constructor in action

Code:

```
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
has_int yet_other = other_int;
yet_other.printme();
```

Output:

```
set: 5
copy: 5
I have: 5
I have: 5
copy: 5
I have: 5
```

59. Copying is recursive

Class with a vector:

```
class has_vector {
private:
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << '\n'; };
};
```

Copying is recursive, so the copy has its own vector:

Code:

```
has_vector a_vector(5);
has_vector other_vector(a_vector);
a_vector.set(3);
a_vector.printme();
other_vector.printme();
```

Output:

```
I have: 3
I have: 5
```

60. Destructor

- Every class `myclass` has a *destructor* `~myclass` defined by default.
- The default destructor does nothing:

```
~myclass() {};
```

- A destructor is called when the object goes out of scope. Great way to prevent memory leaks: dynamic data can be released in the destructor. Also: closing files.

61. Destructor example

Just for tracing, constructor and destructor do `cout`:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << '\n';
  };
  ~SomeObject() {
    cout << "calling the destructor"
         << '\n';
  };
};
```

62. Destructor example

Destructor called implicitly:

Code:

```
cout << "Before the nested scope"
     << '\n';
{
  SomeObject obj;
  cout << "Inside the nested scope"
       << '\n';
}
cout << "After the nested scope"
     << '\n';
```

Output:

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Headers

63. C headers plusplus

You know how to use `.h` files in C.

Classes in C++ need some extra syntax.

64. Data members in proto

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

```
class something {
private:
  int localvar;
public:
  // declaration:
  double somedo(vector);
};
```

Implementation file:

```
// definition
double something::somedo(vector v) {
   .... something with v ....
   .... something with localvar ....
};
```

65. Static class members

A static member acts as if it's shared between all objects.
(Note: C++17 syntax)

Code:

```
class myclass {
private:
  static inline int count=0;
public:
  myclass() { count++; };
  int create_count() {
    return count; };
};
  /* ... */
  myclass obj1,obj2;
  cout << "I have defined "
       << obj1.create_count()
       << " objects" << '\n';
```

Output:

```
I have defined 2 objects
```

66. Static class members, C++11 syntax

```
class myclass {
private:
  static int count;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
// in main program
int myclass::count=0;
```

Class relations: has-a

67. Has-a relationship

A class usually contains data members. These can be simple types or other classes. This allows you to make structured code.

```
class Person {
  string name;
  ....
};
class Course {
private:
  Person the_instructor;
  int year;
};`
```

This is called the has-a relation:

`Course` has-a `Person`

68. Literal and figurative has-a

A line segment has a starting point and an end point.

A *Segment* class can store those points:

```
class Segment {
private:
  Point
     starting_point,ending_point;
public:
  Point get_the_end_point() {
    return ending_point; };
}
int main() {
  Segment somesegment;
  Point somepoint =

     somesegment.get_the_end_point();
```

or store one and derive the other:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
public:
  Point get_the_end_point() {
    /* some computation
       from the
       starting point */ };
}
```

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

69. Polymorphism in constructors

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

```
class Segment {
private:
 // up to you how to implement!
public:
  Segment( Point start,float length,float angle )
    { .... }
  Segment( Point start,Point end ) { ... }
```

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

Exercise 15

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

70. 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*.

71. General case, special case

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

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

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

72. Inheritance: derived classes

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

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

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

73. Constructors

When you run the special case constructor, usually the general constructor needs to run too. By default the 'default constructor', but usually explicitly invoked:

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

74. 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 16

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?

75. 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:

```
class Base {
public:
  virtual f() { ... };
};
class Deriv : public Base {
public:
  virtual f() override { ... };
};
```

76. Override and base method

Code:

```
class Base {
protected:
  int i;
public:
  Base(int i) : i(i) {};
  virtual int value() { return i; };
};

class Deriv : public Base {
public:
  Deriv(int i) : Base(i) {};
  virtual int value() override {
    int ivalue = Base::value();
    return ivalue*ivalue;
  };
};
```

Output:

```
25
```

77. Friend classes

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

```
/* forward definition: */ class A;
class B {
  // A objects can access B internals:
  friend class A;
private:
  int i;
};
class  A {
public:
  void f(B b) { b.i; }; // friend access
};
```

78. Abstract classes

Special syntax for abstract method:

```
class Base {
public:
  virtual void f() = 0;
};
class Deriv : public Base {
public:
  virtual void f() { ... };
};
```

79. 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'.

Vectors

80. 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`.)

81. 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)
```

82. Short vectors

Short vectors can be created by enumerating their elements:

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

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

Exercise 17

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, containing odd numbers, which are the even values plus 1?

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

83. Range over elements

You can write a range-based for loop, which considers the elements as a collection.

```
vector<float> my_data /* create */;
for ( float e : my_data )
  // statement about element e
for ( auto e : my_data )
  // same, with type deduced by compiler
```

Code:

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

Output:

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

Exercise 18

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 19

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.

84. Range over vector denotation

Code:

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

Output:

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

85. Vector definition

Definition and/or initialization:

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

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

where

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

86. Accessing vector elements

Square bracket notation (zero-based):

Code:

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

Output:

```
4,8
```

With bound checking:

Code:

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

Output:

```
4,8
```

Safer, slower.
(Remember Knuth about optimization.)

87. Vector elements out of bounds

Square bracket notation:

Code:

```
vector<int> numbers = {1,4};
numbers[-1] += 3;
numbers[2] = 8;
cout << numbers[0] << ","
     << numbers[1] << '\n';
```

Output:

```
1,4
```

With bound checking:

Code:

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

Output:

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

Safer, slower.

(Remember Knuth about optimization.)

88. 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:

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

Output:

```
6.6
```

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

89. 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:

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

Output:

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

90. 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 20

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

Which mechanism do you use?

Exercise 21

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?

91. Vector copy

Vectors can be copied just like other datatypes:

Code:

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

Output:

```
3.5,7
```

Note: contents copied, not just pointer.

92. 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`

93. 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. More later.

Dynamic behaviour

94. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

```
vector<int> mydata(5,2);
mydata.push_back(35);
cout << mydata.size() << '\n';
cout << mydata.back();
     << '\n';
```

Output:

```
6
35
```

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

95. 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.)

96. Filling in vector elements

You can push elements into a vector:

```
  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`:

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

97. Filling in vector elements

With subscript:

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

You can also use `new` to allocate*:

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

*Considered bad practice. Do not use.

98. 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

99. 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:

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

Output:

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

100. 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.

101. Vector pass by value example

Code:

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

Output:

```
3.5
```

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

102. Vector pass by reference

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

Code:

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

Output:

```
4.6
```

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

103. 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 22

Revisit exercise 21 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 23

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

104. 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.

```
class named_field {
private:
  vector<double> values;
  string name;
```

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

105. 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)) {
  };
};
```

Less desirable method is creating in the constructor:

```
  named_field( string uname,int n ) {
    name = uname;
    values = vector<double>(n);
  };
```

Multi-dimensional arrays

106. 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.

107. Matrix class

```
class matrix {
private:
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
```

Exercise 24

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

Exercise 25

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:

```
A.set(3.);
cout << "Sum of elements: "
     << A.totalsum() << '\n';
```

Output:

```
Sum of elements: 30
```

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

108. Matrix class; better design

Better idea:

```
class Matrix {
private:
  int rows,cols;
  vector<double> elements;
private:
  Matrix( int m,int n )
  : rows(m),cols(n),
    elements(vector<double>(rows*cols))
    {};
  ...
  double get(int i,int j) {
    return elements.at(i*cols+j);
  }
```

(Old-style solution: use cpp macro)

Exercise 26

In the matrix class of the previous slide, why are `m,n` stored explicitly, and not in the previous case?

Exercise 27

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

Implement matrix-matrix multiplication.

109. 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 28

- 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.

```
         *
        * *
       *   *
      * * * *
     *       *
    * *     * *
   *   *   *   *
  * * * * * * * *
 *               *
* *             * *
```

110. 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 29

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

Other array stuff

111. 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.

112. Random walk exercise

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

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?

113. Span

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

The `span` object has the same `at`, `data`, and `size` methods, and you can iterate over it, but it has no dynamic methods.

Strings

114. String declaration

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

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

(Do not use the C legacy mechanisms.)

115. 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";
```

116. Quotes in strings

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

Code:

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

Output:

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

117. Concatenation

Strings can be *concatenated*:

Code:

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

Output:

```
foo bar: 7
```

118. String indexing

You can query the *size*:

Code:

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

Output:

```
5
```

or use subscripts:

Code:

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

Output:

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

119. Ranging over a string

Same as ranging over vectors.

Range-based for:

Code:

```
cout << "By character: ";
for ( char c : abc )
  cout << c << " ";
cout << '\n';
```

Output:

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

Ranging by index:

Code:

```
string abc = "abc";
cout << "By character: ";
for (int ic=0; ic<abc.size(); ic++)
  cout << abc[ic] << " ";
cout << '\n';
```

Output:

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

120. Range with reference

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

Code:

```
for ( char &c : abc )
  c += 1;
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 30

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

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

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

121. More vector methods

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

Code:

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

Output:

```
0
5
```

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

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

Exercise 31

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 32

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

122. String stream

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

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

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

123. String an object, 1

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

```
  string as_string() {
    stringstream ss;
    ss << "(" << x << "," << y << ")";
    return ss.str();
  };
  /* ... */
std::ostream& operator<<
    (std::ostream &out,Point &p) {
  out << p.as_string(); return out;
};
```

124. String an object, 2

Redefine the less-less operator to use this.

```
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << '\n';
```

Exercise 33

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

Code:

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

Output:

```
   1.5
  12.32
 123.456
1234.5678
```

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

I/O

125. Default unformatted output

Code:

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

Output:

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

126. 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.

127. 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>`

128. Fmtlib basics

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

Code:

```
auto hello_string = fmt::format
  ("{} {}!","Hello","world");
cout << hello_string << '\n';
fmt::print
  ("{0}, {0}
     {1}!\n","Hello","world");
```

Output:

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

API documentation: `https://fmt.dev/latest/api.html`

129. Reserve space

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

Code:

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

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

Output:

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

Width is 6:
>     123
```

130. Right aligned in fmtlib

Code:

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{:>6}\n",i);
```

Output:

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

131. Padding character

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

Code:

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

Output:

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

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

132. Left alignment

Instead of right alignment you can do left:

Code:

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

Output:

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

133. Padding characters in fmtlib

Code:

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{0:.>6}\n",i);
```

Output:

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

134. Number base

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

Code:

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

Output:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 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
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
```

135. Number bases in fmtlib

Code:

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

Output:

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

Exercise 34

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
```

136. Fixed point precision

Fixed precision applies to fractional part:

Code:

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

Output:

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

(Notice the rounding)

Exercise 35

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

Code:

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

Output:

```
   1.5
  12.32
 123.456
1234.5678
```

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

137. Scientific notation

Combining width and precision:

Code:

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

Output:

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

138. Text output to file

Use:

Code:

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

Output:

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

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

139. 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.

```
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

140. 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.

141. 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.

142. Example: step 1, we need a class

Simple class that stores one number:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto get() { return x; };
  void set(double xx) { x = xx; };
};
```

143. 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 */ );
```

144. Use of a shared pointer

Same as C-pointer syntax:

Code:

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

  /* ... */
  HasX xobj(5);
  cout << xobj.value() << '\n';
  xobj.set(6);
  cout << xobj.value() << '\n';

  auto xptr = make_shared<HasX>(5);
  cout << xptr->value() << '\n';
  xptr->set(6);
  cout << xptr->value() << '\n';
```

Output:

```
5
6
5
6
```

145. 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;
```

146. Getting the underlying pointer

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

Code:

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

Output:

```
5
6
```

147. Pointers don't go with addresses

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

```
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
```

Exercise 36

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

```
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

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

Code:

```
cout << "Area: " <<  lielow.area()
     << '\n';
/* ... */
// scale the `fivetwo' point by two
cout << "Area: " <<  lielow.area()
     << '\n';
```

Output:

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

You can base this off the file `pointrectangle.cxx` *in the repository*

Automatic memory management

148. 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.

149. Illustration

We need a class with constructor and destructor tracing:

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

150. Show constructor / destructor in actio

Code:

```
cout << "Outside\n";
{
  thing x;
  cout << "create done\n";
}
cout << "back outside\n";
```

Output:

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

151. Illustration 1: pointer overwrite

Let's create a pointer and overwrite it:

Code:

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

Output:

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

152. Illustration 2: pointer copy

Code:

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

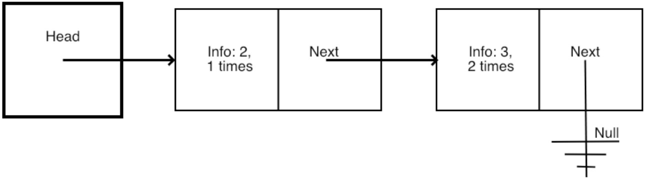
Output:

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

- 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

153. Linked list



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

154. Definition of List class

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

```
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

155. Definition of Node class

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

```
class Node {
private:
  int datavalue{0},datacount{0};
  shared_ptr<Node> next{nullptr};
public:
  Node() {}
  Node(int value,shared_ptr<Node> next=nullptr)
    : datavalue(value),datacount(1),next(next) {};
  int value() {
    return datavalue; };
  auto nextnode() {
    return next; };
```

A Null pointer indicates the tail of the list.

156. 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';
```

157. Print a list

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

Print the list head:

```
void print() {
  cout << "List:";
  if (has_list())
    cout << " => ";
    head->print();
  cout << '\n';
};
```

Print a node and its tail:

```
void print() {
  cout << datavalue << ":" <<
    datacount;
  if (has_next()) {
    cout << ", ";
    next->print();
  }
};
```

158. Recursive length computation

For the list:

```
int recursive_length() {
  if (!has_list())
    return 0;
  else
    return head->listlength();
};
```

For a node:

```
int listlength_recursive() {
  if (!has_next()) return 1;
  else return 1+next->listlength_recursive();
};
```

159. Iterative computation of the list length

Use a shared pointer to go down the list:

```
int length_iterative() {
  int count = 0;
  if (has_list()) {
    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?)

160. 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.

161. Definition of List class

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

```
class List {
private:
  unique_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

162. Definition of Node class

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

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

A Null pointer indicates the tail of the list.

163. Iterative computation of the list length

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

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

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

164. 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

165. 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
};
```

166. 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:

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Output:

```
3.8
```

167. 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.
- Inputs: like function parameters
- Result type specification `->` *outtype*: can be omitted if compiler can deduce it;
- Definition: function body.

168. 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:

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

169. Direct invocation, 2

OpenMP:

```
const int nthreads = [] () -> int {
  int nt;
#pragma omp parallel
#pragma omp master
  nt = omp_get_num_threads();
  return nt;
}();
```

170. Assign lambda expression to variable

Code:

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

Output:

```
3.8
8.9
```

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

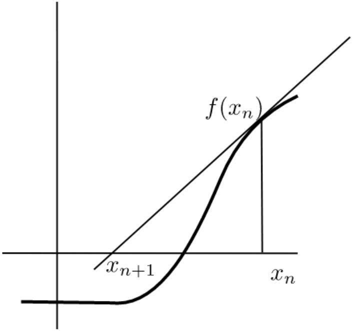
Return type could have been omitted:

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

Example of lambda usage: Newton's method

171. Newton's method

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



172. 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 37

The Newton method (see HPC book) for finding the zero of a function f, that is, finding the x for which $f(x) = 0$, can be programmed by supplying the function and its derivative:

```
double f(double x) { return x*x-2; };
double fprime(double x) { return 2*x; };
```

and the algorithm:

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

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

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

173. Function pointers

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

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

174. Lambdas as parameter: the problem

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

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

(Actually, this simple case does work with C syntax, but not for general lambdas)

175. Lambdas as parameter: the solution

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

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

Code:

```
void apply_to_5
    ( function< void(int) > f ) {
  f(5);
}
  /* ... */
  apply_to_5
    ( [] (int i) {
        cout << "Int: " << i << '\n';
      } );
```

Output:

```
Int: 5
```

176. Lambdas expressions for Newton

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

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

```
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 38

Rewrite the Newton exercise above to use a function that is used as:

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

Call the function

1. first with the lambda variables you already created;
2. but in a better variant, directly with the lambda expressions as arguments, that is, without assigning them to variables.

Captures

177. Capture parameter

Capture value and reduce number of arguments:

```
int exponent=5;
auto powerfive =
  [exponent] (float x) -> float {
    return pow(x,exponent); };
```

Now `powerfive` is a function of one argument, which computes that argument to a fixed power.

Code:

```
cout << "To the power "
     << exponent << '\n';
for (float x=1.; x<=5.; x+=1.)
  cout << x << ":" << powerfive(x)
     << '\n';
```

Output:

```
To the power 5
1:1
2:32
3:243
4:1024
5:3125
```

178. 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 39

- 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?)

Exercise 40

Extend the newton exercise to compute roots in a loop:

```
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.

Exercise 41

You don't need the gradient as an explicit function: you can approximate it as

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

for some value of h.

Write a version of the root finding function

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

that uses this. 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.

More lambda topics

179. Capture by value

Normal capture is by value:

Code:

```
int one=1;
auto increment_by_1 =
  [one] ( int input ) -> int {
    return input+one;
};
cout << increment_by_1 (5)  << '\n';
cout << increment_by_1 (12)  <<
     '\n';
cout << increment_by_1 (25)  <<
     '\n';
```

Output:

```
6
13
26
```

180. 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.)

181. Lambdas vs function pointers

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

Code:

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

Output:

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

182. Use in algorithms

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

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

See later.

Union-like things

Tuples

183. Example for this lecture

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

184. Returning two things

Simple solution:

```
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = 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

185. Function returning tuple

How do you return two things of different types?

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

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

```

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

186. Returning tuple with type deduction

Return type deduction:

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

Alternative:

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

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

187. Catching a returned tuple

The calling code is particularly elegant:

Code:

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

Output:

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

This is known as structured binding.

188. 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

189. 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>
```

```
optional<float> MaybeRoot(float x) {
  if (x<0)
    return {};
  else
    return 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';
```

190. Create optional

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

optional<float> f {
  if (something) return 3.14;
  else return {};
}
```

Expected (C++23)

191. 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

192. 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.

193. Variant methods

```
variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

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

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

(Takes pointer to variant, returns pointer to value)

Exercise 42

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:

```
for ( auto coefficients :
      { make_tuple(2.0, 1.5, 2.5),
        make_tuple(1.0, 4.0, 4.0),
        make_tuple(2.2, 5.1, 2.5)
       } ) {
  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

194. Begin and end iterator

Use independent of looping:

Code:

```
vector<int> v{1,3,5,7};
auto pointer = v.begin();
cout << "we start at "
     << *pointer << '\n';
pointer++;
cout << "after increment: "
     << *pointer << '\n';

pointer = v.end();
cout << "end is not a valid element: "
     << *pointer << '\n';
pointer--;
cout << "last element: "
     << *pointer << '\n';
```

Output:

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

195. Erase at/between iterators

Erase from start to before-end:

Code:

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

Output:

```
1,4
```

(Also single element without end iterator.)

196. Insert at iterator

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

Code:

```
vector<int> counts{1,2,3,4,5,6},
  zeros{0,0};
auto after_one = zeros.begin()+1;
zeros.insert
  ( after_one,
    counts.begin()+1,
    counts.begin()+3 );
cout << zeros[0] << ","
     << zeros[1] << ","
     << zeros[2] << ","
     << zeros[3]
     << '\n';
```

Output:

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

197. Iterator arithmetic

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

Algorithms with iterators

198. Reduction operation

Default is sum reduction:

Code:

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

Output:

```
sum: 16
```

199. Reduction with supplied operator

Supply multiply operator:

Code:

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

Output:

```
product: 30
```

200. Custom reduction function

```
class x {
public:
  int i,j;
  x() {};
  x(int i,int j) : i(i),j(j)
    {};
};
```

```
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

201. 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;
}
```

202. 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 `++`.

203. Custom iterators, 1

Ranging over a class with iterator subclass

Class:

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

};
```

Main:

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

204. Custom iterators, 2

Random-access iterator:

```
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 43

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

```
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); };
```

C++20 ranges

205. Iterate without iterators

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

206. Stream composition

Code:

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << '\n';
auto times_two_over_five = v
  | transform( [] (int i) {
      return 2*i; } )
  | filter( [] (int i) {
      return i>5; } );
cout << "Times two over five: "
  << vector_as_string
      ( times_two_over_five |
      ranges::to_vector )
     << '\n';
```

Output:

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

Namespaces

207. You have already seen namespaces

Safest:

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

Drastic:

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

Prudent:

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

208. Defining a namespace

You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
|
```

209. Namespace usage

Qualify type with namespace:

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

or

```
using namespace abc = space_a::space_b::space_c;
abc::func(x)
```

210. Including and using a namespace

There is a *vector* in the standard namespace and in the new *geometry* namespace:

```
#include <vector>
#include "geolib.h"
using namespace geometry;
int main() {
  std::vector< vector > vectors;
  vectors.push_back( vector( point(1,1),point(4,5) ) );
```

211. Header definition

```
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

212. Implementations

```
namespace geometry {
  point::point( double x,double y ) {
      xcoord = x; ycoord = y; };
  double point::x() { return xcoord; }; // `accessor'
  double point::y() { return ycoord; };
  vector::vector( point from,point to) {
    this->from = from; this->to = to;
  };
  double vector::size() {
    double
      dx = to.x()-from.x(), dy = to.y()-from.y();
    return sqrt( dx*dx + dy*dy );
  };
}
```

213. Why not 'using namespace std'?

This compiles, but should not:

```
#include <iostream>
using namespace std;

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

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

This gives an error:

```
#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;
}
```

Templates

214. Templated type name

If you have multiple routines that do 'the same' for multiple types, you want the type name to be a variable. Syntax:

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

215. Example: function

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 44

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

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

Code:

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

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

Output:

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

216. Templated vector

The Standard Template Library (STL) contains in effect

```
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

217. Exception throwing

Throwing an *exception* is one way of signaling an error or unexpected behavior:

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

218. Catching an exception

It now becomes possible to detect this unexpected behavior by *catching* the exception:

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

Exercise 45

Revisit the prime generator class (exercise 69) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section **??**.)

Code:

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

Output:

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

219. 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
}
```

220. Catch any exception

Catch exceptions without specifying the type:

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

221. 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!

222. 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
  }
```

223. 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 **??**.
- 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`.

224. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

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

Output:

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

225. 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 result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

226. 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. );
```

227. Use assertions during development

Assertions are disabled by

```
#define NDEBUG
```

before the include.

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

Auto

228. 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();
```

229. Type deduction in functions

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

230. Type deduction in methods

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

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

231. Auto and references, 1

`auto` discards references and such:

Code:

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output:

```
data: 5.7
```

232. Auto and references, 2

Combine `auto` and references:

Code:

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

Output:

```
data: 6.7
```

233. Auto and references, 3

For good measure:

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

Casts

234. C++ casts

- `reinterpret_cast`: Old-style 'take this byte and pretend it is XYZ'; very dangerous.
  Instead:
- `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

235. Static cast

```
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:

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

Output:

236. Pointer to base class

Class and derived:

```
class Base {
public:
  virtual void print() = 0;
};
class Derived : public Base {
public:
  virtual void print() {
    cout << "Construct derived!"
         << '\n'; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Construct erived!"
         << '\n'; };
};
```

237. Cast to derived class

This is how to do it:

Code:

```
Base *object = new Derived();
f(object);
Base *nobject = new Erived();
f(nobject);
```

Output:

```
make[1]: Nothing to be
    done for
    `deriveright'.
```

238. Cast to derived class, the wrong way

Do not use this function g:

Code:

```
void g( Base *obj ) {
  Derived *der =
    static_cast<Derived*>(obj);
  der->print();
};
  /* ... */
    Base *object = new Derived();
    g(object);
    Base *nobject = new Erived();
    g(nobject);
```

Output:

```
make[1]: Nothing to be
      done for
      `derivewrong'.
```

Const

239. 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).

240. Constant arguments

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

```
void f(const int i) {
  i++;
}
```

241. Const ref parameters

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

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

242. 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 { thru += i; }
}
```

243. Const polymorphism

A const method and its non-const variant are different enough that you can use this for overloading.

Code:

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

Output:

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

Exercise 46

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

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

244. 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;
}
```

245. 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.)

246. Mutable example

Code:

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

Output:

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

More STL

Complex

247. 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);
```

248. Example usage

Code:

```
vector< complex<double> > vec1(N,
      1.+2.5i );
auto vec2( vec1 );
/* ... */
for ( int i=0; i<vec1.size(); i++ )
      {
   vec2[i] = vec1[i] * ( 1.+1.i );
}
/* ... */
auto sum = accumulate
   ( vec2.begin(),vec2.end(),
     complex<double>(0.) );
cout << "result: " << sum << '\n';
```

Output:

```
result:
      (-1.5e+06,3.5e+06)
```

Limits

249. Templated functions for limits

Use header file `limits`:

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

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

250. Some limit values

Code:

```
cout << "Signed int: "
     << numeric_limits<int>::min()
     << " "
     << numeric_limits<int>::max()
     << '\n';
cout << "Unsigned    "
     << numeric_limits<unsigned int>::min() << " "
     << numeric_limits<unsigned int>::max()
     << '\n';
cout << "Single      "
     << numeric_limits<float>::denorm_min()
     << " "
     << numeric_limits<float>::min()
     << " "
     << numeric_limits<float>::max()
     << '\n';
cout << "Double      "
     << numeric_limits<double>::denorm_min()
     << " "
```

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
```

251. 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:

```
cout << "Single lowest "
     <<
     numeric_limits<float>::lowest()
     << " and epsilon "
     <<
     numeric_limits<float>::epsilon()
     << '\n';
cout << "Double lowest "
     <<
     numeric_limits<double>::lowest()
     << " and epsilon "
     <<
     numeric_limits<double>::epsilon()
     << '\n';
```

Output:

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

Random numbers

252. Random floats

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// 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.);

cout << "first rand: " << distribution(generator) << '\n';
```

253. 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

254. 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"
```

255. Date

coming in C++20

File system

256. file system

```
#include <filesystem>
```

including directory walker

Regular expressions

257. Example

Code:

```
auto cap = regex("[A-Z][a-z]+");
for ( auto n :
      {"Victor", "aDam", "DoD"}
      ) {
  auto match =
    regex_match( n, cap );
  cout << n;
  if (match) cout << ": yes";
  else       cout << ": no" ;
  cout << '\n';
}
```

Output:

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

C++20 modules

258. Modules

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

Unit testing

259. 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 ...

Intro to testing

260. 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.

261. 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

262. 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

263. 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.

264. Unit testing frameworks

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

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

265. Toy example

Function and tester:

```
double f(int n) { return n*n+1; }

#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

TEST_CASE( "test that f always returns positive" ) {
  int n=5;
  REQUIRE( f(n)>0 );
}
```

266. Compiling toy example

```
icpc -o tdd tdd.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

- Files:

  ```
  icpc -o tdd tdd.cxx
  ```

- Path to include and library files:

  ```
  -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB}
  ```

- Libraries:

  ```
  -lCatch2Main -lCatch2
  ```

Make a script file!

267. Correctness through 'require' clause

Tests go in `tester.cxx`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- *TEST_CASE* acts like independent main program.
  can have multiple cases in a tester file
- *REQUIRE* is like `assert` but more sophisticated

268. Tests

Boolean:

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

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Beware floating point:

```
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!=Catch::Approx(1.0) );
```

In general exact tests don't work.

269. Output for failing tests

Run the tester:

```
-------------------------------
test the increment function
-------------------------------
test.cxx:25
...............................

test.cxx:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2

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

270. 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( "function fails for " << n );
    REQUIRE( f(n)>0 );
}
```

Exercise 47: Positive tests

Continue with the example of slide **??**:
add a positive TEST_CASE

```
for (int i=1; i<10; i++)
  REQUIRE( increment_positive_only(i)==i+1 );
```

Make the function satisfy this test.

271. Test for exceptions

Suppose function g(n)

- succeeds for input $n > 0$
- 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) );
}
```

Exercise 48: Negative tests

Make sure your function throws an exception at illegal inputs:

```
for (int i=0; i>-10; i--)
  REQUIRE_THROWS( increment_positive_only(i) );
```

272. Tests with code in common

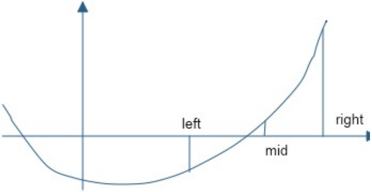
Use SECTION if tests have intro/outtro in common:

```
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)

TDD example: Bisection

273. Root finding by bisection



- Start with bounds where the function has opposite signs.

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

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

274. Coefficient handling

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

We implement this by storing the coefficients in a `vector<double>`.

Proper:

```
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 49: One test for properness

Write a function *is_proper_polynomial* as described, and write unit tests for it, both passing and failing:

```
vector<double> good = /* proper coefficients */ ;
REQUIRE( is_proper_polynomial(good) );
vector<double> notso = /* improper coefficients */ ;
REQUIRE( not is_proper_polynomial(good) );
```

275. Handy shortcut

Are you getting tired of typing `vector<double>`?
put

```
using polynomial = vector<double>;
```

somewhere high in your file.

276. Test on polynomials evaluation

Next we need to evaluate polynomials.

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

```
polynomial second( {2,0,1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

Exercise 50: Implementation

Write a function *evaluate_at* which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up Horner's rule and implement it.

277. Odd degree polynomials only

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

```
if ( not is_odd(coefficients) ) {
  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 51: Odd degree testing

Implement the *is_odd* test.

Gain confidence by unit testing:

```
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) );
```

278. 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?)

```
void find_initial_bounds
  ( polynomial coefficients,double &left,double &right);
```

Since we reject even degree polynomials,
throw an exception for those.

Exercise 52: Test for initial bounds

Unit test:

```
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 );
```

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

279. Move the bounds closer

Root finding iteratively moves the initial bounds closer together:

```
move_bounds_closer(coefficients,left,right);
```

- on input, `left`<`right`, and
- on output the same must hold.

Design a test for this function;
implement this function.

280. Putting it all together

Ultimately we need a top level function

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

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

Exercise 53: Put it all together

Make this call work:

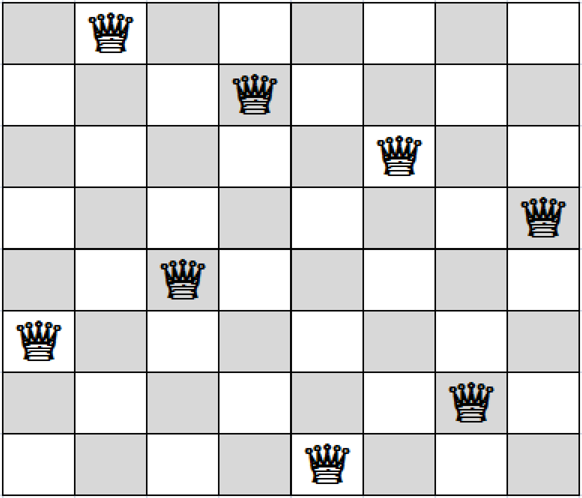
```
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

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

Eight queens problem by TDD (using objects)

281. Problem statement

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

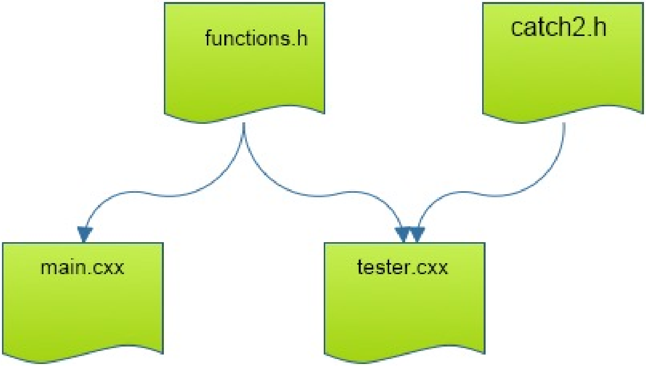


282. Sort of test-driven development

You will solve the ‘eight queens’ problem by

- designing tests for the functionality
- then implementing it

283. File structure



284. Basic object design

Object constructor of an empty board:

```
ChessBoard(int n);
```

Test how far we are:

```
int next_row_to_be_filled()
```

First test:

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

Exercise 54: Board object

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

Exercise 55: Board method

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

```
void place_next_queen_at_column(int i);
```

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

```
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Exercise 56: Test for collisions

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

```
bool feasible()
```

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

```
ChessBoard empty(n);
REQUIRE( empty.feasible() );

ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );

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 57: Test full solutions

Make a second constructor to 'create' solutions:

```
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Now we test small solutions:

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

Exercise 58: No more delay: the hard stuff!

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

```
optional<ChessBoard> place_queens()
```

Test that the last step works:

```
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 59: Test that you can find solutions

Test that there are no 3×3 solutions:

```
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:

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

History of C++ standards

285. 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.

286. C++11

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

287. 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.

288. C++17

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

289. 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 **??**.

290. C++23

- *md_span*