C++ for C Programmers

Victor Eijkhout

TACC/XSEDE training 2022

Code of Conduct

XSEDE has an external code of conduct for XSEDE sponsored events which represents XSEDE's commitment to providing an inclusive and harassment-free environment in all interactions regardless of gender, sexual orientation, disability, physical appearance, race, or religion. The code of conduct extends to all XSEDE-sponsored events, services, and interactions.

Code of Conduct: `https://www.xsede.org/codeofconduct`

Contact:

- Teacher: Victor Eijkhout `eijkhout@tacc.utexas.edu`
- Event organizer: Jason Allison `jasona@tacc.utexas.edu`

XSEDE ombudspersons:

- Linda Akli, Southeastern Universities Research Association `akli@sura.org`
- Lizanne Destefano, Georgia Tech `lizanne.destefano@ceismc.gatech.edu`
- Ken Hackworth, Pittsburgh Supercomputing Center `hackworth@psc.edu`
- Bryan Snead, Texas Advanced Computing Center `jbsnead@tacc.utexas.edu`

1: 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 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. 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.

2: Minor enhancements

6. 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.

7. 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`)

8. 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)

9. 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.

10. Main

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

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

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

3: Functions

11. Big and small changes

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

Parameter passing

12. 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'

13. 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'.

14. 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, see later)

15. 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
[basic] ref:

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

(You will not use references often this way.)

16. Reference create 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’
```

17. 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.
- Reference syntax is cleaner than C 'pass by reference'

18. 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
}
```

19. Pass by reference example 1

Code:

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

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

Output

```
[basic] setbyref:

5
```

Compare the difference with leaving out the reference.

20. 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' ....
```

Exercise 1

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

Code:

```
cout << i << "," << j << "\n";
swap(i,j);
cout << i << "," << j << "\n";
```

Output

```
[func] swap:

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,remainder
  cout << " is divisible by ";
else
  cout << " has remainder "
       << remainder << " from ";
cout << divisor << "\n";
```

Output
[func] divisible:

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

More about functions

21. 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.

22. Polymorphic functions

You can have multiple functions with the same name:

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

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

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

23. 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
```

24. Const ref parameters

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

- Pass by reference: no copying, so cheap
- Const: no accidental altering.

4: Object-Oriented Programming

25. 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.

26. 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
[object] functionality:

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

Note the 'dot' notation.

Exercise 3

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?

27. Construct an object

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

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

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

28. Constructor and data initialization

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

Constructors are typically used to initialize data members.

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

```
Point v(1.,2.);
```

29. Using the default constructor

No constructor explicitly defined:

```
class IamZero {
private:
  int i=0;
public:
  void print() { cout << i; };
};
```

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

```
IamZero zero;
zero.print();
```

30. Default constructor

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()'
```

31. 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.)

32. 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.

33. Food for thought: constructor vs data

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

Example: create a vector from `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.

Methods

34. Functions on objects

Code:

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

Output

```
[geom] pointfunc:

p1 has length 2.23607
```

We call such internal functions 'methods'.
Data members, even `private`, are global to the methods.

35. 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

```
[geom] pointscaleby:

p1 to origin 2.23607
p1 to origin 4.47214
```

36. 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

```
[geom] pointscale:

p1 to origin 2.23607
p2 to origin 4.47214
```

37. 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'

Exercise 4

Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 5

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.

Exercise 6

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

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

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

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

38. 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`

Exercise 7

Write a class `primegenerator` that contains:

- Methods `how_many_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";
}
```

39. 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 really needed.

40. 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 predicates on the members.

41. 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
}
```

42. Const functions

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

43. ‘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!
  };
};
```

44. '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)

45. Operator overloading

Syntax:

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

For instance:

Code:

```
Point Point::operator*(double factor)
    {
  return Point(factor*x,factor*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

```
[geom] pointmult:
p1 to origin 2.23607
scaled right: 4.47214
```

Can also:

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

Exercise 8

Revisit exercise 6 and replace the *add* and *scale* functions by overloaded operators.

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

More constructors

46. 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"; };
};
```

47. Copy constructor in action

Code:

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

Output

```
[object] copyscalar:

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

48. 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.

49. 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";
  };
};
```

50. 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
[object] destructor:

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

Headers

51. C headers plusplus

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

Classes in C++ need some extra syntax.

52. Data members in proto

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

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

Implementation file:

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

53. 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
[link] static17:

```
I have defined 2
    objects
```

54. 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

55. 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*

56. 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.

57. 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 9

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

58. Examples for base and derived cases

- Base case: employee. Has: salary, employee number.
  Special case: manager. Has in addition: underlings.
- Base case: shape in drawing program. Has: extent, area, drawing routine.
  Special case: square et cetera; has specific drawing routine.

59. 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 = ... };
};
```

60. 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.

61. 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) {};
};
```

62. 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 10

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?

63. 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 { ... };
};
```

64. 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

```
[object] virtual:

25
```

65. Friend classes

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

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

66. Abstract classes

Special syntax for abstract method:

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

67. 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'.

5: Vectors

68. Vectors are better than 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`.)

69. 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;
}
```

70. Range over elements

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

```
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

```
[array] dynamicmax:

Max: 6 (should be 6)
```

Exercise 11

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.

71. 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

```
[array] vectorrangeref:

6.6
```

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

72. 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.

73. 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

```
[array] assignbracket:

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

```
[array] assignatfun:

4,8
```

Safer, slower.

(Remember Knuth about optimization.)

74. 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)
```

Exercise 12

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?

75. 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

```
[array] vectorcopy:

3.5,7
```

76. 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`

Dynamic behaviour

77. 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

```
[array] vectorend:

6
35
```

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

78. 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;
```

79. 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.

80. 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
```

81. 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.

82. 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.

Vectors and functions

83. 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

```
[array] vectorreturn:

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

84. 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.

85. 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

```
[array] vectorpassnot:

3.5
```

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

86. 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

```
[array] vectorpassref:

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?

87. 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!)

Vectors in classes

88. 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.

89. 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 nelements )
    : name(name),
      values(vector<double>(n)) {
  };
};
```

Less desirable method is creating in the constructor:

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

90. 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.

91. 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);
  };
```

92. 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 13

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

Implement matrix-matrix multiplication.

93. 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 14

- 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.

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

The object needs to have an array internally. The easiest solution is

94. 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?

6: Strings

95. String declaration

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

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

(Do not use the C legacy mechanisms.)

96. 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";
```

97. 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

```
[string] stringadd:

foo bar: 7
```

98. String indexing

You can query the *size*:

Code:

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

Output

```
[string] stringsize:

5
```

or use subscripts:

Code:

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

Output

```
[string] stringsub:

char three: 2
char four : 3
```

99. 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
[string] stringpush:

```
0
5
```

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

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

100. 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;
```

101. 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;
};
```

102. String an object, 2

Redefine the less-less operator to use this.

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

7: I/O

103. Default unformatted output

Code:

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

Output
[io] cunformat:

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

104. 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.

105. 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>`

106. 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
[io] fmtbasic:

Hello world!
Hello, Hello world!

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

107. 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

```
[io] width:

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
```

108. Right aligned in fmtlib

Code:

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

Output

```
[io] fmtwidth:

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

109. 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
[io] formatpad:

```
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.

110. 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
[io] formatleft:

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

111. Padding characters in fmtlib

Code:

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

Output
[io] fmtleftpad:

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

112. 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
[io] format16:

```
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
```

113. Number bases in fmtlib

Code:

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

Output
[io] fmtbase:

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

Exercise 15

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
```

114. 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
[io] fix:

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

(Notice the rounding)

Exercise 16

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

```
[io] quasifix:

   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.

115. 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

```
[io] iofsci:

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
```

116. Text output to file

The *iostream* is just one example of a stream:
general mechanism for converting entities to exportable form.
In particular: file output works the same as screen output.

Use:

Code:

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

Output
[io] fio:

```
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'.

117. 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";
```

8: Smart pointers

118. 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.

Smart pointers

119. Simple example

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; };
};
```

120. 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

```
[pointer] pointx:

5
6
5
6
```

121. Creating a pointer

Allocation and pointer in one:

```
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
  // or simpler:
auto X = make_shared<Obj>( /* args */ );
```

122. 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

```
[pointer] pointy:

5
6
```

123. 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
```

Automatic memory management

124. Memory leaks

- Vectors obey scope: deallocated automatically.
- Destructor called when object goes out of scope, including exceptions.
- 'RAII'
- Dynamic allocation doesn't obey scope: objects with smart pointers get de-allocated when no one points at them anymore.
  (Reference counting)

125. Illustration

We need a class with constructor and destructor tracing:

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

126. 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

```
[pointer] ptr1:

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

127. 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

```
[pointer] ptr2:
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.

Smart pointer example: linked lists

128. Linked list structures

Linked list: data structure with easy insertion and deletion of information.

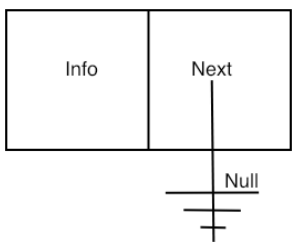
Two basic elements:

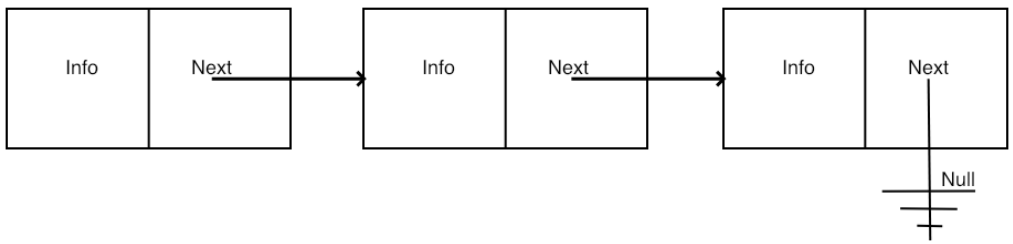
- List, has pointer to first element, or null pointer
- Node, has information, plus pointer to next element (or null)

We are going to look at info routines about a list ('length'), or routines that alter the list ('insert').

129. (in pictures)

Node data structure and linked list of nodes





130. 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.

131. 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};
  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; };
  int count() {
    return datacount; };
  int listlength() {
    if (!has_next()) return 1;
    else return 1+next->listlength();
  };
  bool has_next() {
    return next!=nullptr; };
```

A Null pointer indicates the tail of the list.

132. Recursive length computation

For the list:

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

For a node:

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

133. Iterative computation of the list length

Use a shared pointer to go down the list:

```
int length_iterative() {
  int count = 0;
  auto current_node = head;
  while (current_node!=nullptr) {
    current_node = current_node->next; count += 1;
  }
  return count;
};
```

134. 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 nuther story
- Potentially cheaper because no reference counting.

135. 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.

136. 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.

137. Iterative computation of the list length

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

```
int length_iterative() {
  int count = 0;
  auto current_node = head;
  while (current_node!=nullptr) {
    current_node = current_node->next; count += 1;
  }
  return count;
};
```

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

Advanced pointer topics

138. Opaque pointer

Use `std::any` instead of void pointers.

Code:

```
std::any a = 1;
cout << a.type().name() << ": "
     << std::any_cast<int>(a) << "\n";
a = 3.14;
cout << a.type().name() << ": "
     << std::any_cast<double>(a) <<
     "\n";
a = true;
cout << a.type().name() << ": "
     << std::any_cast<bool>(a) <<
     "\n";

try {
  a = 1;
  cout << std::any_cast<float>(a) <<
     "\n";
} catch (const std::bad_any_cast& e) {
  cout << e.what() << "\n";
}
```

Output

```
[pointer] any:

i: 1
d: 3.14
b: true
bad any cast
```

139. Null pointer

C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

```
void f(int);
void f(int*);
  f(NULL);    // calls the int version
  f(nullptr); // calls the ptr version
```

Note: dereferencing is undefined behaviour; does not throw an exception.

9: Namespaces

140. 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;
}
```

141. Defining a namespace

You can make your own namespace by writing

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

142. 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)
```

143. 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) ) );
```

144. 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();
  };
}
```

145. 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 );
  };
}
```

146. 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;
}
```

10: Templates

147. 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 ...
```

148. 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 17

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

```
[template] eps:

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

149. 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
}
```

11: Exceptions

150. Exception throwing

Throwing an *exception* is one way of signalling an error or unexpected behaviour:

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

151. Catching an exception

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

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

Exercise 18

Revisit the prime generator class (exercise 7) 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

```
[primes] genx:

9931
9941
9949
9967
9973
Reached max int
```

152. 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
}
```

153. Catch any exception

Catch exceptions without specifying the type:

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

154. 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!

155. 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
  }
```

156. 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`.

157. 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

```
[object] exceptdestruct:
calling the
    constructor
Inside the nested
    scope
calling the
    destructor
Exception caught
```

158. 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 );
```

159. 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. );
```

160. Use assertions during development

Assertions are disabled by

```
#define NDEBUG
```

before the include.

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

12: Iterators

161. 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

```
[stl] iter:

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

(Note: the auto actually stands for `vector::iterator`)

162. 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; // easier
    than 'iterator'
counts.erase(second,fourth);
cout << counts[0] << "," << counts[1]
    << "\n";
```

Output

```
[iter] erase2:

1,4
```

(Also single element without end iterator.)

163. 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
     );
//vector<int>::insert(
     after_one,counts.begin()+1,counts.begin()+3
     );
cout << zeros[0] << "," << zeros[1]
     << ","
     << zeros[2] << "," << zeros[3]
     << "\n";
```

Output

```
[iter] insert2:

0,2,3,0
```

164. Reduction operation

Default is sum reduction:

Code:

```
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

```
[stl] accumulate:

sum: 16
```

165. Reduction with supplied operator

Supply multiply operator:

Code:

```
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

```
[stl] product:

product: 30
```

166. Vector iterator

Range-based iteration is syntact sugar around iterator use:

```
for (auto elt_itr=vec.begin(); elt_itr!=vec.end(); ++elt_itr) {
  element = *elt_itr;
  cout << element;
}
```

compare

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

167. Auto iterators

```
vector<int> myvector(20);
for ( auto copy_of_int :
    myvector )
  s += copy_of_int;
for ( auto &ref_to_int :
    myvector )
  ref_to_int = s;
for ( const auto&
    copy_of_thing : myvector )
  s += copy_of_thing.f();
```

is actually short for:

```
for ( std::vector<int>
     iterator
    it=myvector.begin() ;
     it!=myvector.end() ; ++it
    )
   s += *it ; // note the deref
```

Range iterators can be used with anything that is iteratable (vector, map, your own classes!)

168. Iterating backward

Reverse iteration can not be done with range-based syntax.

Use general syntax with reverse iterator: `rbegin`, `rend`.

169. Iterator arithmetic

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

170. Simple illustration

Let's make a class, called a bag, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```
class bag {
  // basic data
private:
  int first,last;
public:
  bag(int first,int last) : first(first),last(last) {};
```

171. Use case

We can iterate over our own class:

Code:

```
bag digits(0,9);

bool find3{false};
for ( auto seek : digits )
  find3 = find3 || (seek==3);
cout << "found 3: " << boolalpha
     << find3 << "\n";

bool find15{false};
for ( auto seek : digits )
  find15 = find15 || (seek==15);
cout << "found 15: " << boolalpha
     << find15 << "\n";
```

Output

```
[loop] bagfind:

found 3: true
found 15: false
```

(for this particular case, use `std::any_of`)

172. With algorithms

Code:

```
bool find8 = any_of
  ( digits.begin(),digits.end(),
    [=] (int i) { return i==8; } );
cout << "found 8: " << boolalpha
     << find8 << "\n";
```

Output

```
[loop] bagany:

found 8: true
```

173. Vector iterator

Range-based iteration is syntact sugar around iterator use:

```
for (auto elt_itr=vec.begin(); elt_itr!=vec.end(); ++elt_itr) {
  element = *elt_itr;
  cout << element;
}
```

compare

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

174. Requirements

- a method `iteratable iteratable::begin()`: initial state
- a method `iteratable iteratable::end()`: final state
- an increment operator `void iteratable::operator++`: advance
- a test `bool iteratable::operator!=(const iteratable&)`
- a dereference operator `iteratable::operator*`: return state

175. Internal state

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```
private:
  int seek{0};
```

176. Initial/final state

The `begin` method gives a `bag` with the seek parameter initialized:

```
public:
  bag &begin()  {
    seek = first; return *this;
  };
  bag end() {
    seek = last; return *this;
  };
```

These routines are public because they are (implicitly) called by the client code.

177. Termination test

The termination test method is called on the iterator, comparing it to the end object:

```
bool operator!=( const bag &test ) const {
  return seek<=test.last;
};
```

178. Dereference

Finally, we need the increment method and the dereference. Both access the `seek` member:

```
void operator++() { seek++; };
int operator*() { return seek; };
```

Exercise 19

Make a `primes` class that can be ranged:

Code:

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << "\n";
```

Output

```
[primes] range:

2, 3, 5, 7, 11, 13,
    17, 19, 23, 29,
    31, 37, 41, 43,
    47, 53, 59, 61,
    67, 71, 73, 79,
    83, 89, 97, 101,
```

13: Auto

179. 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();
```

180. Type deduction in functions

Return type can be deduced in C++17:

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

181. 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"; };
};
```

182. 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

```
[auto] plainget:

data: 5.7
```

183. 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

```
[auto] refget:

data: 6.7
```

184. 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();
```

14: Lambdas

185. A simple example

You can define a function and apply it:

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

or you can apply the function recipe directly:

Code:

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

Output

```
[func] lambdadirect:

3.8
```

186. 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.

187. Assign lambda to variable

Code:

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

Output

```
[func] lambdavar:

3.8
```

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

Return type could have been ommitted:

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

Exercise 20

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 g.

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

188. Lambdas as parameter: the problem

Lambdas are in a class that is dynamically generated, so you can not write a function that takes a lambda as argument, because you don't have the class name.

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

(Do not use C-style function pointer syntax.)

189. Lambdas as parameter: the solution

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

With this, you can declare parameters by their signature:

```
double find_zero
   ( function< double(double) > f,
     function< double(double) > fprime ) {
```

This states that `f` is in the class of `double(double)` functions.

Exercise 21

Rewrite the Newton exercise above to use a function with prototype

```
double root = find_zero( f,g );
```

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.

190. 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

```
[func] lambdait:

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

Exercise 22

Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; n++) {
  cout << "sqrt(" << n << ") = "
       << find_zero(
/* ... */
                    )
       << "\n";
```

Without lambdas, you would define a function

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

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

Exercise 23

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 find_zero( 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 `find_zero` you coded earlier.

191. Capture by value

Normal capture is by value:

Code:

```
int one=1;
auto one_more =
  [one] ( int input ) -> void {
      cout << input+one << "\n";
};
one_more (5);
one_more (6);
one_more (7);
```

Output

```
[func] lambdavalue:

6
7
8
```

192. Capture by value/reference

Capture by reference:

Code:

```
int stride = 1;
auto more_and_more =
  [&stride] ( int input ) -> void {
    cout << input << "=>" <<
     input+stride << "\n";
    stride++;
};
more_and_more(5);
more_and_more(6);
more_and_more(7);
more_and_more(8);
more_and_more(9);
cout << "stride is now: " << stride
     << "\n";
```

Output

```
[func] lambdareference:

5=>6
6=>8
7=>10
8=>12
9=>14
stride is now: 6
```

193. Capture a reduction variable

This mechanism is useful

```
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;
```

194. Lambdas vs 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

```
[func] lambdacptr:

C ptr: 6
Lambda: 6
```

15: Lambda in algorithms

195. For each, very simple example

Apply something to each array element:

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
for_each( ints.begin(),ints.end(),
          [] ( int i ) -> void {
            cout << i << "\n";
          }
          );
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

196. For any

See if any element satisfies a boolean test:

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
bool there_was_an_8 =
  any_of( ints.begin(),ints.end(),
          [] ( int i ) -> bool {
            return i==8;
          }
          );
cout << "There was an 8: " <<
 boolalpha << there_was_an_8 <<
 "\n";
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

197. Capture by reference

Capture variables are normally by value, use ampersand for reference. This is often used in *algorithm* header.

Code:

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count
   << "\n";
```

Output

```
[stl] counteach:

number of even: 3
```

198. For each, with capture

Capture by reference, to update with the array elements.

Code:

```
vector<int>
 ints{2,3,4,5,7,8,13,14,15};
int sum=0;
for_each( ints.begin(),ints.end(),
          [&sum] ( int i ) ->
 void {
            sum += i;
          }
          );
cout << "Sum = " << sum << "\n";
```

Output

```
[iter] each:

2
3
4
5
7
8
13
14
15
```

199. Sorting

Iterator syntax:
(see later for ranges)

```
sort( myvec.begin(),myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(),myvec.end(),
      [] (int i,int j) { return i>j; }
    );
```

16: Casts

200. 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

201. 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

```
[cast] intlong:
```

202. 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"; };
};
```

203. Cast to derived class

This is how to do it:

Code:

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

Output

```
[cast] deriveright:

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

204. 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

```
[cast] derivewrong:

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

17: Tuples

205. 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.

206. Function returning tuple

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)};
};
```

207. 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
[stl] tuple:

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

This is known as structured binding.

208. 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

209. Tuple solution

```
tuple<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,sqrt(x)};
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
      cout << "Root is " << root << "\n";
    else
      cout << "could not take root of " << x << "\n";
```

18: Variants

210. Variant

```
#include <variant>

variant<int,double,string> union_ids;

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";
```

Exercise 24

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

```
[union] quadratic:

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
```

211. Optional results (C++17)

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";
```

Exercise 25

Write a function *first_factor* that optionally returns the smallest factor of a given input.

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << "\n";
```

19: Const

212. 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).

213. 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++;
}
```

214. Const ref parameters

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

- Pass by reference: no copying, so cheap
- Const: no accidental altering.

215. 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; }
}
```

216. 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] constat:

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

Exercise 26

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?

217. 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;
}
```

218. 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.)

20: More STL

Complex

219. 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);
```

220. 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

```
[complex] vec:

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

Limits

221. Templated functions for limits

Use header file `limits`:

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

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

222. 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

```
[stl] limits:

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
```

223. 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
[stl] eps:

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

Random numbers

224. 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";
```

225. 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

226. 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"
```

227. Date

coming in C++20

File system

228. file system

```
#include <filesystem>
```

including directory walker

Regular expressions

229. Example

Code:

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

Output

```
[regexp] regexp:

Looks like a name:
libc++abi.dylib:
     terminating with
     uncaught
     exception of
     type
     std::__1::regex_error:
     Unknown error
     type
make[2]: ***
     [run_regexp]
     Abort trap: 6
```

21: Unit testing

22: Unit testing and test-driven development (TDD)

230. 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 . . .

231. 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.

232. 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

233. 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

234. 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.

235. Unit testing frameworks

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

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

236. 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" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

237. 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
  ```

238. 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 program.
- REQUIRE is like assert but more sophisticated
- Can contain (multiple) tests for correctness.

239. 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.

240. 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
```

241. 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 27: 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.

242. Test for exceptions

Supppose 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 28: 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) );
```

243. Tests with code in common

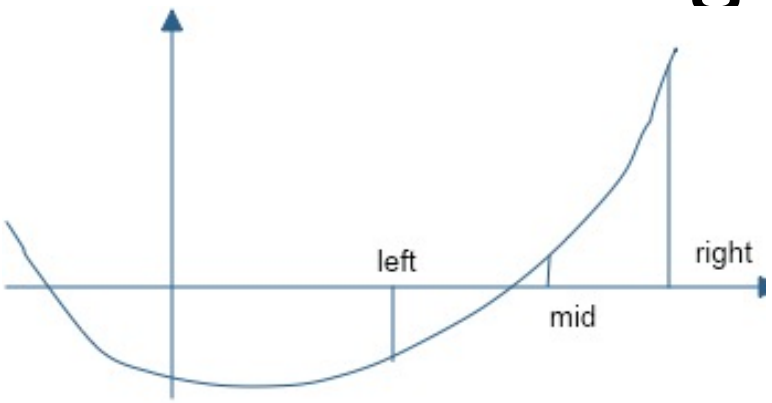
Use SECTION if tests have intro/outtro in common:

```
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)

23: TDD example: Bisection

244. 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.

245. Coefficient handling

$$f(x) = c_d x^d + \cdots + c_1 x^1 + c_0$$

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

```
TEST_CASE( "coefficients are polynomial","[1]" ) {
  auto coefficients = set_coefficients();
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 29: Proper polynomials

Write a routine *set_coefficients* that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Exercise 30: One test for properness

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

246. Test on polynomials evaluation

```
// correct interpretation: 2x^2 + 1
vector<double> second{2,0,1};
REQUIRE( proper_polynomial(second) );
REQUIRE( evaluate_at(second,2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( evaluate_at(second,2) != Catch::Approx(6) );
```

Exercise 31: Implementation

Write a function `evaluate_at` which computes

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

and confirm that it passes the above tests.

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

Exercise 32: Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
Reject even degree polynomials:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

Gain confidence by unit testing:

```
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Exercise 33: Find bounds

Write a function *find_outer* 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?)
Unit test:

```
right = left+1;
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
```

How would you test the function values?

Exercise 34: Put it all together

Make this call work:

```
auto zero = find_zero(coefficients,left,right);
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << "\n";
```

Add an optional precision argument to the root finding function.

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

24: C++20 ranges

247. Iterate without iterators

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

248. Transform and filter

Code:

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

Output

```
[range] ft1:

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

249. 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

```
[range] ft2:

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

250. Adapters

Adaptors take a range and return a view. They can do that by themselves, or chained:

```
auto v = std::views::reverse(vec);
auto v = vec | std::views::reverse;
auto v = vec | std::views::reverse | /* more adaptors */ ;
```

25: C++20 modules

251. Modules

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

26: History of C++ standards

252. 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.

253. C++11

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

254. 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.

255. C++17

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

256. 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 **??**.

257. C++23

- *md_span*