C++ for C Programmers

Victor Eijkhout

TACC Training 2018

Introduction

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.

In this course

1. Object-oriented programming.
2. New mechanisms that replace old ones:
   I/O, strings, arrays, pointers.
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!

About this course

Slides and codes are from my open source text book:

`https://bitbucket.org/VictorEijkhout/`
`textbook-introduction-to-scientific-programming`

Minor enhancements

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
```

- Loop variable can be local:

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

Simple I/O

Headers:

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

Ouput:

```
int main() {
  int OC=4;
  cout << "Hello world (ABEND CODE OC" << OC << ")" << endl;
```

Input:

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

C standard header files

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

But a number of headers are not needed anymore.

Functions

Big and small changes

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

Parameter passing

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'

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 also called ‘input’, ‘output’, ‘throughput’.

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 << endl;
i *= 2;
cout << i << "," << ri << endl;
ri -= 3;
cout << i << "," << ri << endl;
```

Output [basic] ref:

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

(You will not use references often this way.)

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
}
```

Different from C

- C mechanism passes address by value.
- If you find yourself writing asterisks, you're not writing C++.

Pass by reference example 1

Code:

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

  int var = 0;
  f(var);
  cout << var << endl;
```

Output [basic] setbyref:

```
5
```

Compare the difference with leaving out the reference.

Pass by reference example 2

```
bool can_read_value( int &value ) {
  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;
```

Exercise 1

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

```
int i=2,j=3;
swapij(i,j);
// now i==3 and j==2
```

Optional exercise 2

Write a function that tests divisibility and returns a remainder:

```
int number,divisor,remainder;
// read in the number and divisor
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

More about functions

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.

Polymorphic functions

You can have multiple functions with the same name:

```
double sum(double a,double b) {
  return a+b; }
double sum(double a,double b,double c) {
  return a+b+c; }
```

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

Const parameters

You can prevent local changes to the function parameter:

```
/* This does not compile:
   void change_const_scalar(const int i) { i += 1; }
*/
```

This is mostly to protect you against yourself.

Object-Oriented Programming

Classes look a bit like structures

Code:

```
class Vector {
public:
  double x,y;
};

int main() {
  Vector p1;
  p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.
  cout << "sum of components: " << p1.x+p1.y  << endl;
```

Output [geom] pointstruct:

```
sum of components: 3
```

Class definition versus object declaration.
We'll get to that '`public`' in a minute.

Class initialization and use

Use a *constructor*: function with same name as the class.

```
class Vector {
private: // recommended!
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
}; // end of class definition


Vector p1(1.,2.);
```

Example of accessor functions

Getting and setting of members values is done through accessor functions:

```
class Vector {
private: // recommended!
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
}; // end of class definition
```

```
public:
  double x() { return vx; };
  double y() { return vy; };
  void setx( double newx ) {
    vx = newx; };
  void sety( double newy ) {
    vy = newy; };
```

```
Vector p1(1.,2.);
```

Usage:

```
p1.setx(3.12);
/* ILLEGAL: p1.x() = 5; */
cout << "P1's x=" << p1.x() << endl;
```

Public versus private

- Implementation: data members, keep `private`,
- Interface: `public` functions to get/set data.
- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

Private access gone wrong

We make a class with two members that sum to one.
You don't want to be able to change just one of them!

```
class SumIsOne {
public:
  float x,y;
  SumIsOne( double xx ) { x = xx; y = 1-x; };
}
int main() {
  SumIsOne pointfive(.5);
  pointfive.y = .6;
}
```

In general: enforce predicates on the members.

Member default values

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

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

Each object will have its members initialized to these values.

Member initialization

Other syntax for initialization:

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

Allows for reuse of names:

Code:

```
class Vector {
private:
  double x,y;
public:
  Vector( double x,double y ) : x(x),y(y) {
  }
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 = "
       << p1.getx() << "," << p1.gety()
       << endl;
```

Output [geom] pointinitxy:

```
p1 = 1,2
```

'this'

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 is not often needed. Typical use case: you need to call a function inside a method that needs the object as argument)

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

Methods

Functions on objects

Code:

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
  double length() { return sqrt(vx*vx + vy*vy); };
  double angle() { return 0.; /* something trig */; };
};

int main() {
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
```

Output [geom] pointfunc:

```
p1 has length 2.23607
```

We call such internal functions 'methods'.

Data members, even `private`, are global to the methods.

Methods that alter the object

Code:

```
class Vector {
  /* ... */
  void scaleby( double a ) {
    vx *= a; vy *= a; };
  /* ... */
};
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 has length " << p1.length() << endl;
```

Output [geom] pointscaleby:

```
p1 has length 2.23607
p1 has length 4.47214
```

Methods that create a new object

Code:

```
class Vector {
  /* ... */
  Vector scale( double a ) {
    return Vector( vx*a, vy*a ); };
  /* ... */
};
  /* ... */
  cout << "p1 has length " << p1.length() << endl;
  Vector p2 = p1.scale(2.);
  cout << "p2 has length " << p2.length() << endl;
```

Output [geom] pointscale:

```
p1 has length 2.23607
p2 has length 4.47214
```

Default constructor

```
Vector p1(1.,2.), p2;
cout << "p1 has length " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 has length " << p2.length() << endl;
```

gives (g++; different for intel):

```
pointdefault.cxx: In function 'int main()':
pointdefault.cxx:32:21: error: no matching function for call to
                'Vector::Vector()'
   Vector p1(1.,2.), p2;
```

The problem is with p2. How is it created? We need to define two constructors:

```
Vector() {};
Vector( double x,double y ) {
  vx = x; vy = y;
};
```

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 3

Write a class `primegenerator` that contains

- members `number_of_primes_found` and `last_number_tested`,
- a method `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" << endl;
}
```

Direct alteration of internals

Return a reference to a private member:

```
class Vector {
private:
  double vx,vy;
public:
  double &x() { return vx; };
};
int main() {
  Vector v;
  v.x() = 3.1;
}
```

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() {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

More constructors

Copy constructor

- Several default copy constructors are defined
- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed, for instance for deep copy.

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

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
```

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.

Destructor example

Destructor called implicitly:

Code:

```
class SomeObject {
public:
  SomeObject() { cout <<
    "calling the constructor"
    << endl; };
  ~SomeObject() { cout <<
    "calling the destructor"
    << endl; };
};
  /* ... */
  cout << "Before the nested scope" << endl;
  {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
  }
  cout << "After the nested scope" << endl;
```

Output [object] destructor:

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

Destructors and exceptions

The destructor is called when you throw an exception:

Code:

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

Output [object] exceptobj:

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

Headers

C headers plusplus

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

Classes in C++ need some extra syntax.

Class prototypes

Header file:

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

Implementation file:

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

Strangely, data members also go in the header file.

Static class members

A static member acts like shared between all objects.

```
class MyClass {
private:
  static object_count;
public:
  MyClass() { object_count++; };
}
```

Initialization has to be done elsewhere:

```
MyClass::object_count = 0;
```

Class relations: has-a

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

This is called the *has-a relation*.

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; };
}
  ...
  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.

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 bothering the user.

Exercise 4

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

  ```
  Rectangle(Point bl,float w,float h);
  ```

  The logical implementation is to store these quantities. Implement methods

  ```
  float area(); float width(); float height();
  ```

- Add a second constructor

  ```
  Rectangle(Point bl,Point tr);
  ```

  Can you figure out how to use initializer lists for passing the points?

- Rewrite your class so that it stores two `Point` objects.

Class inheritance: is-a

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 = ... };
};

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

Inheritance: derived classes

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

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

Members and methods need to be `protected`, not `private`, to be inheritable.

Constructors

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

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

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 (see section **??**).

Exercise 5

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?

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 { ... };
};
```

Operator overloading

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

For instance:

```
class Point {
private:
  float x,y;
public:
  Point operator*(float factor) {
    return Point(factor*x,factor*y);
  };
};
```

Can even redefine equals and parentheses.

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'.
- Friend classes:

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

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

Arrays

General note about syntax

Many of the examples in this lecture need the compiler option `-std=c++11`. This works for both compilers, so:

```
// for Intel:
icpc -std=c++11 yourprogram.cxx
// for gcc:
g++ -std=c++11 yourprogram.cxx
```

Later examples with `auto` even need `-std=c++17`.
There is no reason not to use that all the time.

Static arrays

Array creation

New syntax for creation:

```
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << endl;
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << endl;
}
```

Range over elements

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

```
for ( float e : array )
  // statement about element with value e
for ( auto e : array )
  // same, with type deduced by compiler
```

Code:

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output [array] dynamicmax:

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

Range over elements by reference

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

Code:

```
vector<int> numbers = {1,4,2,6,5};
for ( auto &v : numbers )
  v *= 3;
cout << "Scale 0'th by 3: " << numbers[0] << endl;
```

Output [array] dynamicscale:

```
Scale 0'th by 3: 3
```

Vectors

Vector definition

Definition, mostly without initialization.

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

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

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.

Accessing vector elements

You have already seen the square bracket notation:

```
vector<double> x(5, 0.1 );
x[1] = 3.14;
cout << x[2];
```

Alteratively:

```
x.at(1) = 3.14;
cout << x.at(2);
```

Safer, slower.

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 = new double[n]; // please don't
```

Ranging over a vector

```
for ( auto e : my_vector)
  cout << e;
```

Note that e is a copy of the array element:

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output [array] vectorrangecopy:

```
3.3
```

Ranging over a vector by reference

To set array elements, make e 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[2] << endl;
```

Output [array] vectorrangeref:

```
6.6
```

Vector initialization

You can initialize a vector as a whole:

```
vector<int> odd_array{1,3,5,7,9};
vector<int> even_array = {0,2,4,6,8};
```

(This syntax requires compilation with the `-std=c++11` option.)

Vector initialization'

There is a syntax for initializing a vector with a constant:

```
vector<float> x(25,3.15);
```

which gives a vector of size 25, with all elements initialized to 3.15.

Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v[2] = 3.5;
vcopy = v;
cout << vcopy[2] << endl;
```

Output [array] vectorcopy:

```
./vectorcopy
3.5
```

Vector methods

- Get elements with `ar[3]` (zero-based indexing).
  (for C programmers: this is not dereferencing, this uses an operator method)
- Get elements, including bound checking, with `ar.at(3)`.
- Size: `ar.size()`.
- Other functions: `front`, `back`.

Vector indexing

Your choice: fast but unsafe, or slower but safe

```
vector<double> x(5);
x[5] = 1.; // will probably work
x.at(5) = 1.; // runtime error!
```

Dynamic extension

Extend with `push_back`:

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output [array] vectorend:

```
6
35
```

also `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

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

Vector of vectors.

Dynamic behaviour

Dynamic size extending

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

Vector extension

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;
```

Vector extension

With subscript:

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

You can also use `new` to allocate (see section **??**):

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

Timing

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

Vectors and functions

Vector as function return

You can have a vector as return type of a function:

Code:

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

Output [array] vectorreturn:

```
./vectorreturn
x1 size: 10
zero element check: 10
```

Vector as function argument

You can pass a vector to a function:

```
void print0( vector<double> v ) {
  cout << v[0] << endl;
};
```

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

Vector pass by value example

Code:

```
void set0
  ( vector<float> v,float x )
{
  v[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output [array] vectorpassnot:

```
./vectorpassnot
3.5
```

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[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output [array] vectorpassref:

```
./vectorpassref
4.6
```

Vectors in classes

Can you make a class around a vector?

Vector needs to be created with the object, so you can not have the size in the class definition

```
class witharray {
private:
  vector<int> the_array( ???? );
public:
  witharray( int n ) {
    thearray( ???? n ???? );
  }
}
```

Create and assign

The following mechanism works:

```
class witharray {
private:
  vector<int> the_array;
public:
  witharray( int n ) {
    thearray = vector<int>(n);
  }
}
```

Matrix class

```
class matrix {
private:
  int rows,cols;
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    rows = m; cols = 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);
  };
};
```

Matrix class'

Better idea:

```
elements = vector<double>(rows*cols);
...
void get(int i,int j) {
  return elements.at(i*cols+j);
}
```

Exercise 6

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

Implement matrix-matrix multiplication.

Strings

String declaration

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

// .. and now you can use ‘string’
```

(Do not use the C legacy mechanisms.)

String creation

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable (use `-std=c++11`), or assign it dynamically:

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

Concatenation

Strings can be *concatenated*:

```
txt = txt1+txt2;
txt += txt3;
```

String is like vector

You can query the *size*:

```
int txtlen = txt.size();
```

or use subscripts:

```
cout << "The second character is <<" <<
        txt[1] << ">>" << endl;
```

More vector methods

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

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

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

Smart pointers

Pointers and references

C and F pointers

C++ and Fortran have a clean reference/pointer concept: a reference or pointer is an 'alias' of the original object

C/C++ also has a very basic pointer concept:
a pointer is the address of some object
(including pointers)

If you're writing C++ you should not use it.
if you write C, you'd better understand it.

Reference: change argument

```
void f( int &i ) { i += 1; };
int main() {
  int i = 2;
  f(i); // makes it 3
```

Reference: save on copying

```
class BigDude {
private:
   vector<double> array(5000000);
}
int main() {
   BigDude big;
   f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) { .... };
```

Smart pointers

Creating a shared pointer

Allocation and pointer in one:

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

X->method_or_member;
```

Pointers don't go with addresses

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

```
auto
  p1 = shared_ptr<HasY>( &y ),
  p2 = shared_ptr<HasY>( &y );
p1->y = 3;
cout << "Pointer 2's y: "
     << p2->y << endl;
```

gives:

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

Simple example

Code:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
```

Output [pointer] pointx:

```
5
6
```

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 << endl;
Y.get()->y = 6;
cout << ( *Y.get() ).y << endl;
```

Output [pointer] pointy:

```
5
6
```

Pointers don't go with addresses

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

```
auto
  p1 = shared_ptr<HasY>( &y ),
  p2 = shared_ptr<HasY>( &y );
p1->y = 3;
cout << "Pointer 2's y: "
     << p2->y << endl;
```

gives:

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

Automatic memory management

Memory leaks

- Vectors obey scope: deallocated automatically.
- Stuff in objects get destructed when the object is destructed:
- Vectors in objects prevent memory leaks!
- Destructor called when object goes out of scope, including exceptions.
- 'RAII'

Simple example

Code:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
```

Output [pointer] pointx:

```
5
6
```

Reference counting illustrated

We need a class with constructor and destructor tracing:

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

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

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

Output [pointer] ptr1:

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

Pointer copy

Code:

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

Output [pointer] ptr2:

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

Linked list code

```
node *node::prepend_or_append(node *other) {
  if (other->value>this->value) {
    this->tail = other;
    return this;
  } else {
    other->tail = this;
    return other;
  }
};
```

Can we do this with shared pointers?

A problem with shared pointers

```
shared_pointer<node> node:prepend_or_append
    ( shared_ptr<node> other ) {
  if (other->value>this->value) {
    this->tail = other;
```

So far so good. However, `this` is a `node*`, not a `shared_ptr<node>`, so

```
    return this;
```

returns the wrong type.

Solution: shared from this

It is possible to have a 'shared pointer to this' if you define your node class with (warning, major magic alert):

```
class node : public enable_shared_from_this<node> {
```

This allows you to write:

```
    return this->shared_from_this();
```

Namespaces

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;
}
```

Why not 'using namespace std'?

This compiles, but should not:

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

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

This gives an error:

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

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

Big namespace no-no

Do not put `using` in a header file that a user may include.

Defining a namespace

You can make your own namespace by writing

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

Namespace usage

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

or

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

or

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

Templates

Templated type name

Basically, you want the type name to be a variable. Syntax:

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

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 7

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:

```
float float_eps;
epsilon(float_eps);
cout << "For float, epsilon is " << float_eps << endl;

double double_eps;
epsilon(double_eps);
cout << "For double, epsilon is " << double_eps << endl;
```

Templated vector

the Standard Template Library (STL) contains in effect

```
template<typename T>
class vector {
private:
  // data definitions omitted
public:
  T at(int i) { /* return element i */ };
  int size() { /* return size of data */ };
  // much more
}
```

Exceptions

Exception throwing

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

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

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;
}
```

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!

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
}
```

Catch any exception

Catch exceptions without specifying the type:

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

Exceptions in constructors

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

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

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 { ... };
```

Destructors and exceptions

The destructor is called when you throw an exception:

Code:

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

Output [object] exceptobj:

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

Iterators

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;

// short for:

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

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

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) {};
```

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 << endl;

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

Output [loop] bagfind:

```
found 3: true
found 15: false
```

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

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};
```

Initial/final state

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

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

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

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;
};
```

Dereference

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

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

Exercise 8

Make a `primes` class that can be ranged:

Code:

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

Output [primes] range:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

I/O

Basic formatting

Code:

```
#include <iomanip>
using std::setfill;
using std::setw;
//codesnippet formatpad
  /* ... */

int main() {

  //codesnippet formatpad
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.') << setw(6) << i
 << endl;
```

Output [io] formatpad:

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

Basic formatting

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 << endl;
  }
```

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 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Streams

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

Auto

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) );
```

Type deduction in functions

Return type can be deduced in C++17:

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

Type deduction in functions

Return type 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 << endl; };
};
```

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
```

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
```

Auto and references, 3

For good measure:

Code:

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

Output [auto] constrefget:

```
make[2]: *** No rule to make target ‘error_constrefget.c
```

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;

// short for:

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

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

Lambdas

Lambda expressions

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

Example:

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

Store lambda in a variable:

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

Capture parameter

Capture value and reduce number of arguments:

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

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

Lambda in object

```
#include <functional>
using std::function;
  /* ... */
class SelectedInts {
private:
  vector<int> bag;
  function< bool(int) > selector;
public:
  SelectedInts( function< bool(int) > f ) {
    selector = f; };
  void add(int i) {
    if (selector(i))
      bag.push_back(i);
  };
  int size() { return bag.size(); };
};
```

Illustration

```
SelectedInts greaterthan
  ( [threshold] (int i) -> bool { return i>threshold; } );
for (int i=0; i<upperbound; i++)
  greaterthan.add(i);
cout << "Ints under " << upperbound
     << " greater than " << threshold << ": "
     << greaterthan.size() << endl;
```

Background Square roots through Newton

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you have a value y and you want want to compute $x \leftarrow \sqrt{y}$. This is equivalent to finding the zero of

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

where y is fixed. To indicate this dependence on y, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f_y'(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

Exercise 9

Refer to 175 for background, and note that finding x such that $f(x) = a$ is equivalent to applying Newton to $f(x) - a$.

Implement a class `valuefinder` and its `double find(double)` method.

```
class valuefinder {
private:
  function< double(double) >
      f,fprime;
  double tolerance{.00001};
public:
  valuefinder
  ( function< double(double) > f,
    function< double(double) > fprime )
    : f(f),fprime(fprime) {};
```

used as

```
double root = newton_root.find(number);
```

```
class valuefinder {
private:
  function< double(double) >
      f,fprime;
  double tolerance{.00001};
public:
  valuefinder
  ( function< double(double) > f,
    function< double(double) > fprime )
    : f(f),fprime(fprime) {};
```

Exercise 10

Can you write a derived class `rootfinder` used as

```
squarefinder newton_root;
double root = newton_root.find(number);
```

Casts

C++ casts

Old-style 'take this byte and pretend it is XYZ': `reinterpret_cast`

Casting with classes:

- `static_cast` cast base to derived without check.
- `dynamic_cast` cast base to derived with check.

Adding/removing const: `const_cast`

Syntactically clearly recognizable.

Const cast

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << endl;
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: " << bignumber << endl;
```

Code:

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

Output [cast] intlong:

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

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!" << endl; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Construct erived!" << endl; };
};
```

Pass base pointer:

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

Cast to derived class

This is how to do it:

Code:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to Derived"
 << endl;
  else
    der->print();
};
```

Output [cast] deriveright:

```
Construct derived!
Could not be cast to Derived
```

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();
};
```

Output [cast] derivewrong:

```
Construct derived!
Construct erived!
```