C++ for C Programmers

Victor Eijkhout

0th Edition, 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.
   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.

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

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
  ```

- Many variants of initialization syntax!
- Loop variable can be local:

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

Simple I/O

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  int OC=4;
  cout << "Hello world (ABEND CODE OC" << OC << ")" << endl
```

C standard header files

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

But a number of headers are not needed anymore.

Functions; parameter passing through references

Parameter passing

Mathematical type function

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function.
- *pass by value*

Code:

```
double f( double x ) {
  x = x*x;
  return x;
}
  /* ... */
  number = 5.1;
  cout << "Input starts as: "
       << number << endl;
  other = f(number);
  cout << "Input var is now: "
       << number << endl;
```

Output:

```
Input starts as: 5.1
Input var is now: 5.1
Output var is: 26.01
```

Results other than through return

Also good design:

- Return no function result,
- or return (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’.

Parameter passing by reference

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

Pass by reference example 1

```
void f( int &i ) {
  i = 5;
}
int main() {
  int var = 0;
  f(var);
  cout << var << endl;
```

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
```

Exercise 2

Write a function that tests divisibility and returns a remainder:

```
int number,divisor,remainder;
// get the number and divisor from the user
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 input parameters: can not differ only in return type.

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:

```
./pointstruct
sum of components: 3
```

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;
  };
  double x() { return vx; }; // 'accessor'
  double y() { return vy; };
};

int main() {
  Vector p1(1.,2.);
```

Example of accessor functions

```
double x() { return vx; };
double y() { return vy; };


void setx( double newx ) { vx = newx; };
void sety( double newy ) { vy = newy; };
```

Usage:

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

Interface versus implementation

- 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

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

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) +
  }
```

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:

```
./pointfunc
p1 has length 2.23607
```

We call such internal functions '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:

```
./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:

```
./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;
```

So:

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

Exercise 3

Write a class `primegenerator` that contains the members of the structure, and the functions `nextprime`, `isprime`. The function `nextprime` does not need the object as argument, because the members are in the object, and therefore global to that function.

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;
}
```

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

Compare:

```
class Segment {
private:
  Point starting_point,ending_point;
}
  ...
  Segment somesegment;
  Point somepoint = somesegment.get_the_end_point();
```

Versus:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
}
```

Implementation vs API.

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!

Exercise 4

Make a class `Rectangle` (sides parallel to axes) with two constructors:

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

and functions

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

Let the `Rectangle` object store two `Point` objects.

Then rewrite your exercise so that the `Rectangle` stores only one point (say, lower left), plus the width and height.

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();
```

Inheritance: derived classes

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

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

Data needs 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) {};
};
```

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 *override* a base class method:

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

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'.

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
```

Static arrays

Array creation

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

```
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:

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

Vectors

Vector definition

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

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

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.
- `value` is the uniform initial value of all elements.

Vector elements

In a number of ways, vector behaves like an array:

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

Ranging over a vector

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

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:

```
3.3
```

Ranging over a vector by reference

To set array elements, make e a reference:

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

Code: **Output:**

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

```
6.6
```

Vector initialization

You can initialize a vector with much the same syntax as an array:

```
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 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!
```

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:

```
./vectorcopy
3.5
```

Vector methods

- Get elements with `ar[3]` (zero-based indexing).
- Get elements, including bound checking, with `ar.at(3)`.
- Size: `ar.size()`.
- Other functions: `front`, `back`, `insert`, `erase`.

Dynamic extension

```
vector<int> array(5);
array.push_back(35);
cout << array.size(); // is now 6 !
```

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

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

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;
point = std::chrono::system_clock::now();
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);
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  stat.at(i) = i;
```

Vector extension

With subscript:

```
vector<int> stat(LENGTH);
stat[0] = 0.;
point = std::chrono::system_clock::now();
for (int i=0; i<LENGTH; i++)
  stat[i] = i;
```

You can also use `new` to allocate:

```
int *stat = new int[LENGTH];
point = std::chrono::system_clock::now();
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:

```
./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:

```
./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:

```
./vectorpassref
4.6
```

Vectors in classes

Can you make a class around a vector?

Vector needs to be created with the object:

```
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 namespace std;

// .. 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.

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

Pointers and references

Pointers and addresses

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 ) { .... };
```

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!
- RAII

Shared pointers

```
#include <memory>

auto array = std::shared_ptr<double>( new double[100] );
```

Constructor and destructor tracing

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

Trace1: pointer overwrite

let's create a pointer and overwrite it:

Code:

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

Output:

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

Trace2: pointer copy

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
  shared_ptr<thing>( new 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:

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

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;
}
```

Best:

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

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;
}
```

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
}
```

I/O