Objects and classes

Victor Eijkhout, Susan Lindsey

Fall 2024

`last formatted: September 30, 2024`

Class basics

Classes

1. 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.

2. Running example

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

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

Exercise 1

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

3. Object functionality

Small illustration: point objects.

Code:

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

Output:

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

Note the ‘dot’ notation.

Exercise 2

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?

4. The object workflow

- First define the class, with data and function members:

```
class MyObject {
  // define class members
  // define class methods
};
```

(details later) typically before the *main*.

- You create specific objects with a declaration

```
MyObject
  object1( /* .. */ ),
  object2( /* .. */ );
```

- You let the objects do things:

```
object1.do_this();
x = object2.do_that( /* ... */ );
```

5. Construct an object

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

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

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

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

6. Private and public

Best practice we will use:

```
class MyClass {
private:
  // data members
public:
  // methods
}
```

- Data is private: not visible outside of the objects.
- Methods are public: can be used in the code that uses objects.
- You can have multiple private/public sections, in any order.

Methods

7. Class methods

Definition and use of the *distance* function:

Code:

```
// geom/pointclass.cpp
class Point {
private:
  float x,y;
public:
  Point(float in_x,float
    in_y) {
    x = in_x; y = in_y; };
  float distance_to_origin() {
    return sqrt( x*x + y*y );
  };
};
    /* ... */
  Point p1(1.0,1.0);
  float d =
    p1.distance_to_origin();
  cout << "Distance to
    origin: "
```

Output:

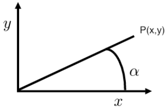
```
Distance to origin:
    ↪1.41421
```

8. Class methods

- Methods look like ordinary functions,
- except that they can use the data members of the class, for instance x, y;
- Methods can only be used on an object with the 'dot' notation. They are not independently defined.

Exercise 3

Add a method `angle` to the `Point` class. How many parameters does it need?



Hint: use the function `atan` or `atan2`.

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

Exercise 4

Make a class *GridPoint* for points that have only integer coordinates. Implement a function *manhattan_distance* which gives the distance to the origin counting how many steps horizontal plus vertical it takes to reach that point.

9. Food for thought: constructor vs data

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

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

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

Note: no change to outward API.

Exercise 5

Discuss the pros and cons of this design:

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

10. Data access in methods

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

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

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

Exercise 6

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.

Quiz 1

T/F?

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

11. Methods that alter the object

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

Code:

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

Output:

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

Data initialization

12. 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.

13. Data initialization

The naive way:

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

The preferred way:

```
// geom/pointinit.cpp
class Point {
private:
  float x,y;
public:
  Point( float in_x,
         float in_y )
    : x(in_x),y(in_y) {
  }
```

Explanation later. It's technical.

Interaction between objects

14. Methods that create a new object

Code:

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

Output:

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

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

15. Anonymous objects

Create a point by scaling another point:

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

Two ways of handling the `return` statement of the `scale` method:

Naive:

```
// geom/pointscale.cpp
Point Point::scale( float a
    ) {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

Creates point, copies it to *new_point*

Better:

```
// geom/pointscale.cpp
Point Point::scale( float a
    ) {
  return Point( x*a, y*a );
};
```

Creates point, moves it directly to *new_point*

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

Exercise 7

Write a method *translated* that, given a *Point* and two *floats*, returns the point translated by those amounts:

Code:

```
// geom/halfway.cpp
Point translated( float
    xt,float yt ) {
  return Point( x+xt,y+yt );
};
```

Output:

```
(1.5,2.5)
by: 0.2,0.3
(1.7,2.8)
```

Optional exercise 8

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?

16. Constructor/destructor

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

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

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

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

17. Using the default constructor

No constructor explicitly defined;

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

Code:

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

Output:

```
1
```

18. Default constructor

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

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

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

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

19. Default constructor

The problem is with *p2*:

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

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

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

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

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

  (but often you can avoid needing it)

Exercise 9

Make a class *LinearFunction* with constructor:

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

where the first stands for a line through the origin.
Implement again the *evaluate* function so that

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

Exercise 10

Can you extend the previous exercise to let

```
LinearFunction line( p1 )
```

mean a line through the origin?

20. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```cpp
// object/stream.cpp
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

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

Output:

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

21. Preliminary to the following exercise

A prime number generator has:
an API of just one function: `nextprime`

To support this it needs to store:
an integer `last_prime_found`

Programming Project Exercise 11

Write a class *primegenerator* that contains:

- Methods *number_of_primes_found* and *nextprime*;
- Also write a function *isprime* that does not need to be in the class.

Your main program should look as follows:

```
// primes/6primesbyclass.cpp
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

Programming Project Exercise 12

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

First formulate the quantor structure of this statement, then translate that top-down to code, using the generator you developed above.

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e,p,q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

22. A Goldbach corollary

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

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

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

In particular this holds for each prime number:

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

We now have the statement that each prime number is the average of two other prime numbers.

Programming Project Exercise 13

Write a program that tests this. You need at least one loop that tests all primes r; for each r you then need to find the primes p, q that are equidistant to it.

Use your prime generator. Do you use two generators for this, or is one enough? Do you need three, for p, q, r?

For each r value, when the program finds the p, q values, print the p, q, r triple and move on to the next r.

Class inclusion: has-a

23. Has-a relationship

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

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

This is called the has-a relation:

`Course` has-a `Person`

24. Literal and figurative has-a

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

Store both points:

```
class Segment {
private:
  Point p_start,p_end;
public:
  Point end_point() {
    return p_end; };
}
int main() {
  Segment seg;
  Point somepoint =
    seg.end_point();
```

or store one and derive the other:

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

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

25. Constructors in has-a case

Class for a person:

```
class Person {
private:
  string name;
public:
  Person( string name ) {
    /* ... */
  };
};
```

Class for a course, which contains a person:

```
class Course {
private:
  Person instructor;
  int enrollment;
public:
  Course( string instr,int n
    ) {
    /* ???? */
  };
};
```

Declare a `Course` variable as: `Course("Eijkhout",65);`

26. Constructors in the has-a case

Possible constructor:

```
Course( string teachername,int nstudents ) {
  instructor = Person(teachername);
  enrollment = nstudents;
};
```

Preferred:

```
Course( string teachername,int nstudents )
  : instructor(Person(teachername)),
    enrollment(nstudents) {
};
```

27. Rectangle class

To implement a rectangle with sides parallel to the x/y axes, two designs are possible. For the function:

```
float Rectangle::area();
```

it is most convenient to store width and height.
For inclusion testing:

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 14

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

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

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

Implement methods:

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

and write a main program to test these.

Exercise 15

Add a second constructor

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

Can you find a way to combine the constructors through constructor delegating? This can be done two ways!

Optional exercise 16

Make a copy of your solution of the previous exercise, and redesign your class so that it stores two `Point` objects. Your main program should not change.

Class inheritance: is-a

28. Hierarchical object relations

Hierarchical relations between classes:

- each object in class A is also in class B.
  but not conversely
- Example: each manager is an employee
- Example: each square is a rectangle

29. Example of class hierarchy

- Class *Employee*:

```
class Employee {
private:
  int number,salary;
/* ... */
};
```

- class *Manager* is subclass of *Employee*
  (every manager is an employee, with number and salary)
- Manager has extra field *n_minions*

How do we implement this?

30. Another example: multiple subclasses

- Example: both triangle and square are polygons.
- You can implement a method `draw` for both triangle/square
- … or write it once for polygon, and then use that.

31. Terminology

Derived classes *inherit* data and methods from the base class: base class data and methods are accessible in objects of the derived class.

- Example: `Polygon` is the *base class*
  `Triangle` is a *derived class*
  Triangle has `corners` because Polygon has
- `Employee` is the *base class*.
  `Manager` is a *derived class*
  Manager has `employee_number` because Employee has

32. Base/Derived example

```
class Polygon {
protected:
vector<Point> corners;
public:
int ncorners() { return corners.size(); };
};
class Triangle : public Polygon {
/* constructor omitted */
};
int main () {
  Triangle t;
  cout << t.ncorners(); // prints 3, we hope
```

33. Examples for base and derived cases

General *FunctionInterpolator* class with method *value_at*. Derived classes:

- *LagranceInterpolator* with *add_point_and_value*;
- *HermiteInterpolator* with *add_point_and_derivative*;
- *SplineInterpolator* with *set_degree*.

34. 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 = ... };
};
```

35. 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.

36. Constructors

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

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

37. 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 17

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?

Exercise 18

Revisit the `LinearFunction` class. Add methods `slope` and `intercept`.

Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

38. 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 { ... };
};
```

39. 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'.

Advanced topics

Operator overloading

40. Better syntax

Operations that 'feel like arithmetic''"

So far:

```
Point p3 = p1.add(p2);
Point p4 = p3.scale(2.5);
```

Improved:

```
Point p3 = p1+p2;
Point p4 = p3*2.5;
```

This is possible because you can *overload* the *operators*. For instance,

```
// geom/overload.cpp
Point operator*( float f ) {
  return Point( x*f,y*f );
}
```

41. Operator overloading

Syntax:

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

For instance:

Code:

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

Output:

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

Exercise 19

Define the plus operator between `Point` objects. The declaration is:

```
Point operator+(Point q);
```

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

Exercise 20

Revisit exercise 8 and replace the `add` and `scale` functions by overloaded operators.

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

42. Functor example

Simple example of overloading parentheses:

Code:

```
// object/functor.cpp
class IntPrintFunctor {
public:
  void operator()(int x) {
    cout << x << '\n';
  }
};
    /* ... */
  IntPrintFunctor intprint;
  intprint(5);
```

Output:

```
5
```

Exercise 21

Evaluate a linear function:

Using method:

```
// geom/overload.cpp
LinearFunction line(p1,p2);
cout << "Value at 4.0: "
     << line.evaluate_at(4.0)
     << '\n';
```

using operator:

```
// geom/overload.cpp
float y = line(4.0);
cout << y << '\n';
```

Write the appropriate overloaded operator.

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

Inherit from containers

43. What is the problem?

You want a `std::vector` but with some added functionality.

```
// proposed construct call:
namedvec<float> x("xvec",5);
// proposed usage:
x.size();
x.name();
x[4];
```

44. Has-a std container

You could write

```
class namedvec {
private:
    std::string name;
    std::vector<float> contents;
public:
    namedvec( std::string n,int s );
    // ...
};
```

The problem now is that for every vector method, `at`,`size`,`push_back`, you have to re-implement that for your `namedvec`.

45. Inherit from vector

Named vector inherits from standard vector:

```cpp
// object/container0.cpp
#include <vector>
#include <string>
class namedvector
  : public std::vector<int> {
private:
  std::string _name;
public:
  namedvector
    ( std::string n,int s )
    : _name(n)
    , std::vector<int>(s) {};
  auto name() {
    return _name; };
};
```

```cpp
// object/container0.cpp
namedvector
    fivevec("five",5);
cout << fivevec.name()
     << ": "
     << fivevec.size()
     << '\n';
cout << "at zero: "
     << fivevec.at(0)
     << '\n';
```

Exercise 22

Extend the code for *namedvector* to make the class templated.

```
// object/container.cpp
namedvector<float> fivetemp("five",5);
cout << fivetemp.name()
     << ": "
     << fivetemp.size() << '\n';
cout << "at zero: "
     << fivetemp.at(0) << '\n';
```

Exercise 23

Extend the code from 78 and 22 to make a namespaced class *geo::vector* that has the functionality of *namedvector*.

```
// object/container.cpp
using namespace geo;
geo::vector<float> float4("four",4);
cout << float4.name() << '\n';
float4[1] = 3.14;
cout << float4.at(1) << '\n';
geo::vector<std::string> string3("three",3);
string3.at(2) = "abc";
cout << string3[2] << '\n';
```

Internal access

46. Direct alteration of internals

Return a reference to a private member:

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

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

47. 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
}
```

48. Access gone wrong

We make a class for points on the unit circle

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

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

In general: enforce invariants on the members.

49. Const functions

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

50. ‘this’ pointer to the current object

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

51. '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)

Headers

52. C headers plusplus

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

Classes in C++ need some extra syntax.

53. Data members in proto

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

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

Implementation file:

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

54. Static class members

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

Code:

```
// link/static17.cpp
class myclass {
private:
  static inline int count=0;
public:
  myclass() { ++count; };
  int create_count() {
    return count; };
};
    /* ... */
  myclass obj1,obj2;
  cout << "I have defined "
       << obj1.create_count()
       << " objects" << '\n';
```

Output:

```
I have defined 2
     ↪objects
```

55. Static class members, C++11 syntax

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