Objects and classes

Victor Eijkhout, Susan Lindsey

Fall 2022
last formatted: August 28, 2022

1: 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:

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

Output
[object] functionality:

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

Note the ‘dot’ notation.

Exercise 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. Constructor and data initialization

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

Constructors are typically used to initialize data members.

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

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

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.

2: Methods

7. Class methods

Let's define method *distance*.

Definition in the class:

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

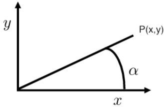
Use in the program:

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

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

Exercise 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.cxx` *in the repository*

Exercise 4

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

8. 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.

9. Food for thought: constructor vs data

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

Example: create a vector from x,y cartesian coordinates, but store *r*,*theta* polar coordinates:

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

Note: no change to outward API.

Exercise 5

Discuss the pros and cons of this design:

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

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.

Review quiz 1

T/F?

- A class is primarily determined by the data it stores.
  `/poll "Class determined by its data" "T" "F"`
- A class is primarily determing by its methods.
  `/poll "Class determined by its methods" "T" "F"`
- If you change the design of the class data, you need to change the constructor call.
  `/poll "Change data, change constructor proto too" "T" "F"`

11. Methods that alter the object

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

Code:

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

Output
[geom] pointscaleby:

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

3: Interaction between objects

12. Methods that create a new object

Code:

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

Output
[geom] pointscale:

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

13. Anonymous objects

Create a point by scaling another point:

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

Two ways of handling the `return` statement:

Naive:

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

Creates point, copies it to `new_point`

Better:

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

Creates point, moves it directly to `new_point`

'move semantics'

Optional exercise 7

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?

14. Default constructor

Consider this code that looks like variable declaration, but for objects:

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

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

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

15. Default constructor

The problem is with *p2*:

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

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

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

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

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

  (but only if you really need it.)

16. Public versus private

- Interface: `public` functions that determine the functionality of the object; effect on data members is secondary.
- Implementation: data members, keep `private`: they only support the functionality.

This separation is a Good Thing:

- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

Exercise 8

Make a class `LinearFunction` with a constructor:

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

and a member function

```
float evaluate_at( float x );
```

which you can use as:

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

17. Classes for abstract objects

Objects can model fairly abstract things:

Code:

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

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

Output
[object] stream:

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

Project Exercise 9

Write a class `primegenerator` that contains:

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

Your main program should look as follows:

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

Turn it in!

- If you have compiled your program, do:

  `coe_primes yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_primes -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with

  `coe_primes -i yourprogram.cc`
- If you don't understand what the script is telling you, try the debug flag:

  `coe_primes -d yourprogram.cc`

Project Exercise 10

The Goldbach conjecture says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to a bound that you read in. Use the `primegenerator` class you developed in exercise 31.

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

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

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

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

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

Turn it in!

- If you have compiled your program, do:

  `coe_goldbach yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.

- Is it reporting that your program is correct? If so, do:

  `coe_goldbach -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.

- If you don't manage to get your code working correctly, you can submit as incomplete with

  `coe_goldbach -i yourprogram.cc`

4: Advanced stuff about constructors

18. Member initializer lists

Other syntax for initialization:
use initializer list.

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

19. Advantage of initializer list

Allows for reuse of names:

Code:

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

Output
[geom] pointinitxy:

```
p1 = 1,2
```

20. Constructors and contained classes

Finally, if a class contains objects of another class,

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

21. Intent checking

Compiler checks your intent against your implementation. This code is not legal:

```
subroutine ArgIn(x)
  implicit none
  real,intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

22. When are contained objects created?

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

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

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

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

23. Copy constructor

- Default defined copy and 'copy assignment' constructors:

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

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

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

24. Copy constructor in action

Code:

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

Output
[object] copyscalar:

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

25. Copying is recursive

Class with a vector:

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

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

Code:

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

Output
[object] copyvector:

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

26. 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.

27. Destructor example

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

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

28. Destructor example

Destructor called implicitly:

Code:

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

Output
[object] destructor:

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

5: Other object stuff

29. String an object

1. Define a function that yields a string representing the object, and
2. redefine the less-less operator to use this.

```
#include <sstream>
using std::stringstream;
```

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

30. Class declarations

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

```
double something::dosomething( int i, char c ) {
  // do something with i,c
};
```