Objects and classes

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2020

Classes

Definition of object

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*.

Objects comes in classes:
similar to variables and datatypes.

When designing a class, first ask yourself: 'what functionality should the objects support'.

Object functionality

Small illustration: vector objects.

Code:

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

Output [object] functionality:

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

Note the 'dot' notation; in a `struct` we use it for the data members; in an object we (also) use it for methods.

Exercise 1

Thought exercise:
What data does the object need to store to do this?
Is there more than one possibility?

The object workflow

Similar to `struct`:

- You have to declare what an object looks like by giving a

  ```
  class myobject { /* ... */ };
  ```

  definition, typically before the `main`.
- You create specific objects with a declaration

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

- You let the objects do things:

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

Constructor

Use a constructor: function with same name as the class.
Typically used to initialize data members.

Straightforward:

```
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.);
```

Preferred:

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

The syntax `x(in_x)` copies the argument to the data member.
(You're even allowed to have `x(x)`.)

Methods

Class methods

Let's define method *length*.

Definition in the class:

```
double length() {
  return sqrt(x*x + y*y); };
```

Use in the program:

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

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

Exercise 2

Add a method `angle` to the `Point` class.

How many parameters does it need?

Exercise 3

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; };
};
```

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 in the constructor

The members stored can be different from the constructor arguments.

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

```
class Point {
private: // members
  double r,theta;
public: // methods
  Point( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan(y/x);
  }
```

Data access in methods

Data members should not be accessed directly from outside an object, but using them inside a method is proper:

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

Interaction between objects

Exercise 4

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.

Hint: remember the 'dot' notation for members.

Review quiz 1

T/F?

- A class is primately 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 class data, you need to change the constructor
  `/poll "Change data, change constructor too" "T" "F"`

Methods that alter the object

Code:

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

Output
[geom] pointscaleby:

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

Methods that create a new object

Code:

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

Output
[geom] pointscale:

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

Anonymous objects

(also known as 'move semantics') Instead of

```
Point scale( double a ) {
  return Point( vx*a, vy*a ); };
```

we could have written:

```
Point scale( double a ) {
  Point new_point( vx*a, vy*a );
  return new_point;
};
```

However, that may involve an extra copy.
(Depends on standard version.)

Optional exercise 5

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

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

Default constructor

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

```
Point p1(1.,2.), p2;
cout << "p1 to origin " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 to origin " << p2.length() << endl;
```

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()'
```

Default constructor

The problem is with *p2*:

```
Point p1(1.,2.), p2;
```

- *p1* is created with the constructor;
- *p2* uses the default constructor;
- 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) {};
```

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 6

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;
```

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() << endl;
  cout << "Next: "
       << ints.next() << endl;
  cout << "Next: "
       << ints.next() << endl;
```

Output
[object] stream:

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

Project Exercise 7

Write a class primegenerator that contains

- members how_many_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;
}
```

Project Exercise 8

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.

This is a great exercise for a top-down approach! Make an outer loop over the even numbers e. In each iteration, make a `primegenerator` object to generate p values. For each p test whether $e - p$ 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_gold 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_gold -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_gold -i yourprogram.cc`

Other object stuff

Class prototypes

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( std::vector<double> v );
};
```

Implementation file:

```
double something::dosomething( std::vector<double> v ) {
  // do something with v
};
```

Advanced stuff about 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; };
};
```

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
```

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) << endl; };
};
```

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
```

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

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

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

Destructor example

Destructor called implicitly:

Code:

```
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
```