

# Objects and classes

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: September 6, 2023

## 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.

```
1 class Point {  
2     /* stuff */  
3 };  
4 int main () {  
5     Point p; /* stuff */  
6 }
```

# 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:

```
1 // /functionality.cpp
2 Point p(1.,2.); // make point
   (1,2)
3 cout << "distance to origin "
4     << p.distance_to_origin()
5     << '\n';
6 p.scaleby(2.);
7 cout << "distance to origin "
8     << p.distance_to_origin()
9     << '\n'
10    << "and angle " << p.angle()
11    << '\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);
```

```
1 class Point {  
2 private: // data members  
3     double x,y;  
4 public: // function members  
5     Point  
6         ( double x_in,double y_in  
7             ) {  
8             x = x_in; y = y_in;  
9         };  
10    /* ... */  
11 };
```

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:

```
1 // /pointclass.cpp
2 class Point {
3 private:
4     float x,y;
5 public:
6     Point(float ux,float uy) { x =
        ux; y = uy; };
7     float distance_to_origin() {
8         return sqrt( x*x + y*y );
9     };
10 };
11     /* ... */
12     Point p1(1.0,1.0);
13     float d = p1.distance_to_origin();
14     cout << "Distance to origin: "
15         << d << '\n';
```

Output:

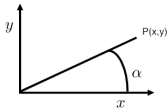
Distance to origin:  
1.41421

## 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.cxx` 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:

```
1 #include <cmath>
2 class Point {
3 private: // members
4     double r, theta;
5 public: // methods
6     Point( double x, double y ) {
7         r = sqrt(x*x+y*y);
8         theta = atan2(y/x);
9     }
```

Note: no change to outward API.



## Exercise 5

Discuss the pros and cons of this design:

```
1 class Point {  
2     private:  
3         double x,y,r,theta;  
4     public:  
5         Point(double xx,double yy) {  
6             x = xx; y = yy;  
7             r = // sqrt something  
8             theta = // something trig  
9         };  
10        double angle() { return alpha; };  
11    };
```

## 10. Data access in methods

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

```
1 class Point {  
2     private:  
3         double x,y;  
4     public:  
5         void flip() {  
6             Point flipped;  
7             flipped.x = y; flipped.y = x;  
8             // more  
9         };  
10 };
```

(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 determined 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:

```
1 // /pointscaleby.cpp
2 class Point {
3     /* ... */
4     void scaleby( double a ) {
5         x *= a; y *= a; };
6     /* ... */
7 };
8     /* ... */
9     Point p1(1.,2.);
10    cout << "p1 to origin "
11         << p1.length() << '\n';
12    p1.scaleby(2.);
13    cout << "p1 to origin "
14         << p1.length() << '\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:

```
1 class Point {  
2 private:  
3     double x,y;  
4 public:  
5     Point( double in_x,  
6           double in_y ) {  
7         x = in_x; y = in_y;  
8     };
```

The preferred way:

```
1 // /pointinit.cpp  
2 class Point {  
3 private:  
4     double x,y;  
5 public:  
6     Point( double userx,  
7           double usery )  
8         : x(userx),y(usery) {  
9     }
```

Explanation later. It's technical.



## Interaction between objects

## 14. Methods that create a new object

Code:

```
1 // /pointscale.cpp
2 class Point {
3     /* ... */
4     Point scale( double a ) {
5         auto scaledpoint =
6             Point( x*a, y*a );
7         return scaledpoint;
8     };
9     /* ... */
10    cout << "p1 to origin "
11          << p1.dist_to_origin()
12          << '\n';
13    Point p2 = p1.scale(2.);
14    cout << "p2 to origin "
15          << p2.dist_to_origin()
16          << '\n';
```

Output:

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

Note the 'anonymous object' in the assignment

# 15. Anonymous objects

Create a point by scaling another point:

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

Two ways of handling the `return` statement:

Naive:

```
1 // /pointscale.cpp
2 Point Point::scale( double a )
3 {
4     Point scaledpoint =
5     Point( x*a, y*a );
6     return scaledpoint;
7 };
```

Creates point, copies it to *new\_point*

Better:

```
1 // /pointscale.cpp
2 Point Point::scale( double a )
3 {
4     return Point( x*a, y*a );
5 };
```

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 *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 with 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:

```
1 // /default.cpp
2 class IamOne {
3 private:
4     int i=1;
5 public:
6     void print() {
7         cout << i << '\n';
8     };
9 };
10 /* ... */
11 IamOne one;
12 one.print();
```

Output:

1

## 18. Default constructor

Refer to point definition: 9

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.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() {};
```

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

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

(but often you can avoid needing it)



## 20. Other way

State that the default constructor exists with the default keyword:

```
// /default.cpp
Point() = default;
Point( double x, double y )
    : x(x), y(y) {};
```

State that there should be no default constructor with the delete keyword:

```
Point() = delete;
```

## 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;
```

# 21. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
1 // /stream.cpp
2 class Stream {
3 private:
4     int last_result{0};
5 public:
6     int next() {
7         return last_result++; };
8 };
9
10 int main() {
11     Stream ints;
12     cout << "Next: "
13         << ints.next() << '\n';
14     cout << "Next: "
15         << ints.next() << '\n';
16     cout << "Next: "
17         << ints.next() << '\n';
```

Output:

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

## 22. 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 9

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:

```
// /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 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.

This is a great exercise for a top-down approach! First formulate the quantor structure of this statement, then translate that to code:

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

## 23. 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:

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

In particular this holds for each prime number:

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

# Programming Project Exercise 11

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. 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  $r$ .



**Advanced stuff**

## 24. Direct alteration of internals

Return a reference to a private member:

```
1 class Point {  
2 private:  
3     double x,y;  
4 public:  
5     double &x_component() { return x; };  
6 };  
7 int main() {  
8     Point v;  
9     v.x_component() = 3.1;  
10 }
```

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

## 25. Reference to internals

Returning a reference saves you on copying.

Prevent unwanted changes by using a 'const reference'.

```
1 class Grid {
2 private:
3     vector<Point> thepoints;
4 public:
5     const vector<Point> &points() const {
6         return thepoints; };
7 };
8 int main() {
9     Grid grid;
10    cout << grid.points()[0];
11    // grid.points()[0] = whatever ILLEGAL
12 }
```

## 26. Access gone wrong

We make a class for points on the unit circle

```
1 // /unit.cpp
2 class UnitCirclePoint {
3 private:
4     float x,y;
5 public:
6     UnitCirclePoint(float x) {
7         setx(x); };
8     void setx(float newx) {
9         x = newx; y = sqrt(1-x*x);
10    };
```

You don't want to be able to change just one of  $x,y$ !  
In general: enforce invariants on the members.

## 27. Const functions

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

## 28. 'this' pointer to the current object

Inside an object, a pointer to the object is available as this:

```
1 class MyClass {
2 private:
3     int myint;
4 public:
5     MyClass(int myint) {
6         this->myint = myint; // `this' redundant!
7     };
8 };
```

## 29. '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)

```
1 /* forward definition: */ class someclass;
2 void somefunction(const someclass &c) {
3     /* ... */ }
4 class someclass {
5     // method:
6     void somemethod() {
7         somefunction(*this);
8     };
```

(Rare use of dereference star)

Operator overloading



# 30. Operator overloading

Syntax:

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

For instance:

Code:

```
1 // /pointscale.cpp
2 Point Point::operator*(double f) {
3     return Point(f*x,f*y);
4 };
5     /* ... */
6 cout << "p1 to origin "
7     << p1.dist_to_origin() <<
8     '\n';
9     Point scale2r = p1*2.;
10    cout << "scaled right: "
11        << scale2r.dist_to_origin()
12        << '\n';
13    // ILLEGAL Point scale2l = 2.*p1;
```

Output:

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

## Exercise 12

Revisit exercise 7 and replace the *add* and *scale* functions by overloaded operators.

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

## 31. Constructors and contained classes

Finally, if a class contains objects of another class,

```
1 class Inner {  
2 public:  
3     Inner(int i) { /* ... */ }  
4 };  
5 class Outer {  
6 private:  
7     Inner contained;  
8 public:  
9 };
```

## 32. 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.

## 33. 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.

```
1 // /copyscalar.cpp  
2 class has_int {  
3 private:  
4     int mine{1};  
5 public:  
6     has_int(int v) {  
7         cout << "set: " << v  
8             << '\n';  
9         mine = v; };  
10    has_int( has_int &h ) {  
11        auto v = h.mine;  
12        cout << "copy: " << v  
13            << '\n';  
14        mine = v; };  
15    void printme() {  
16        cout << "I have: " << mine  
17            << '\n'; };  
18 };
```

## 34. Copy constructor in action

Code:

```
1 // /copyscalar.cpp
2 has_int an_int(5);
3 has_int other_int(an_int);
4 an_int.printme();
5 other_int.printme();
6 has_int yet_other = other_int;
7 yet_other.printme();
```

Output:

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

## 35. Copying is recursive

Class with a vector:

```
1 // /copyvector.cpp
2 class has_vector {
3 private:
4     vector<int> myvector;
5 public:
6     has_vector(int v) { myvector.push_back(v); };
7     void set(int v) { myvector.at(0) = v; };
8     void printme() { cout
9         << "I have: " << myvector.at(0) << '\n'; };
10 };
```

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

Code:

```
1 // /copyvector.cpp
2 has_vector a_vector(5);
3 has_vector other_vector(a_vector);
4 a_vector.set(3);
5 a_vector.printme();
6 other_vector.printme();
```

Output:

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

## 36. 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.



## 37. Destructor example

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

```
1 // /destructor.cpp
2 class SomeObject {
3 public:
4     SomeObject() {
5         cout << "calling the constructor"
6             << '\n';
7     };
8     ~SomeObject() {
9         cout << "calling the destructor"
10            << '\n';
11     };
12 };
```

## 38. Destructor example

Destructor called implicitly:

Code:

```
1 // /destructor.cpp
2 cout << "Before the nested scope"
3     << '\n';
4 {
5     SomeObject obj;
6     cout << "Inside the nested
7         scope"
8         << '\n';
9 }
10 cout << "After the nested scope"
    << '\n';
```

Output:

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

## Headers

## 39. C headers plusplus

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

Classes in C++ need some extra syntax.

## 40. Data members in proto

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

```
1 class something {  
2 private:  
3   int localvar;  
4 public:  
5   // declaration:  
6   double somedo(vector);  
7 };
```

Implementation file:

```
1 // definition  
2 double something::somedo(vector v) {  
3   .... something with v ....  
4   .... something with localvar ....  
5 };
```

# 41. Static class members

A static member acts as if it's shared between all objects.

(Note: C++17 syntax)

Code:

```
1 // /static17.cpp
2 class myclass {
3 private:
4     static inline int count=0;
5 public:
6     myclass() { ++count; };
7     int create_count() {
8         return count; };
9 };
10      /* ... */
11 myclass obj1,obj2;
12 cout << "I have defined "
13      << obj1.create_count()
14      << " objects" << '\n';
```

Output:

*I have defined 2 objects*

## 42. Static class members, C++11 syntax

```
1 // /static.cpp
2 class myclass {
3 private:
4     static int count;
5 public:
6     myclass() { ++count; };
7     int create_count() { return count; };
8 };
9     /* ... */
10 // in main program
11 int myclass::count=0;
```