Objects and classes

Kevin Schmidt, Susan Lindsey, Charlie Dey

Spring 2019

Classes

Classes look a bit like structures

Code:

```
class Vector {
public:
  double x,y;
};

int main() {
  Vector p1;
  p1.x = 1.; // This Is Not A Good Idea
    . See later.
  p1.y = 2.; // same
  cout << "sum of components: "
       << p1.x+p1.y  << endl;
```

Output
[geom] pointstruct:

```
sum of components: 3
```

Class definition versus object declaration.
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;
  };

Vector p1(1.,2.);
```

Example of accessor functions

Getting and setting of members values is done through accessor functions:

```
class Vector {
private: // recommended!
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };

public:
  double x() { return vx; };
  double y() { return vy; };
  void setx( double newx ) {
    vx = newx; };
  void sety( double newy ) {
    vy = newy; };

}; // end of class definition

Vector p1(1.,2.);
```

Usage:

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

Public versus private

- 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

We make a class for points on the unit circle
You don't want to be able to change just one of `x,y`!

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

In general: enforce predicates on the members.

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 initializer lists

Other syntax for initialization:

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

advantages

Allows for reuse of names:

Code:

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

Output
[geom] pointinitxy:

```
p1 = 1,2
```

Also saves on constructer invocation if the member is an object.

Initializer lists

Initializer lists can be used as denotations.

```
Point(float ux,float uy) {
/* ... */
Rectangle(Point bl,Point tr) {
/* ... */
Point origin{0.,0.};
Rectangle lielow( origin, {5,2} );
```

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
[geom] pointfunc:

```
p1 has length 2.23607
```

We call such internal functions 'methods'.

Data members, even `private`, are global to the methods.

Exercise 1

Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `printout` uses cout to display the point.
- `angle` computes the angle of vector (x, y) with the x-axis.

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
[geom] 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
[geom] pointscale:

```
p1 has length 2.23607
p2 has length 4.47214
```

Default constructor

```
Vector v1(1.,2.), v2;
cout << "v1 has length " << v1.length() << endl;
v2 = v1.scale(2.);
cout << "v2 has length " << v2.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 v1(1.,2.), v2;
```

The problem is with `v2`. How is it created? We need to define two constructors:

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

Exercise 2

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.

Exercise 3

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.

Access to internals

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;
  };

Vector p1(1.,2.);
```

Example of accessor functions

Getting and setting of members values is done through accessor functions:

```
class Vector {
private: // recommended!
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };

public:
  double x() { return vx; };
  double y() { return vy; };
  void setx( double newx ) {
    vx = newx; };
  void sety( double newy ) {
    vy = newy; };

}; // end of class definition

Vector p1(1.,2.);
```

Usage:

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

Use accessor functions!

```
class PositiveNumber { /* ... */ }
class Point {
private:
  // data members
public:
  Point( float x,float y ) { /* ... */ };
  Point( PositiveNumber r,float theta ) { /* ... */ };
  float get_x() { /* ... */ };
  float get_y() { /* ... */ };
  float get_r() { /* ... */ };
  float get_theta() { /* ... */ };
};
```

Functionality is independent of implementation.

Exercise 4

Make a class `LinearFunction` with a constructor:

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

and a 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;
```

Exercise 5

Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point input_p2 );
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;
```

Classes for abstact 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 6

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 7

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:

  `sdstestgold yourprogram.cc`

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

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

  `sdstestgold -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

  `sdstestgold -i yourprogram.cc`

More 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

Destructor called implicitly:

Code:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << endl;
  };
  ~SomeObject() {
    cout << "calling the destructor"
         << endl;
  };
};
  /* ... */
  cout << "Before the nested scope"
       << endl;
  {
    SomeObject obj;
    cout << "Inside the nested scope"
         << endl;
```

Output
[object] destructor:

```
make[4]: ./destructor: No such
make[4]: *** [run_destructor] E
```

Other object stuff

Class prototypes

Header file:

```
class something {
public:
  double somedo(vector);
};
```

Implementation file:

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

Strangely, data members also go in the header file.