Objects and classes

Victor Eijkhout and Carrie Arnold and Charlie Dey

Fall 2017

Classes

Classes look a bit like structures

Code:

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

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

Output:

```
./pointstruct
sum of components: 3
```

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;
  };
  double x() { return vx; }; // 'accessor'
  double y() { return vy; };
};

int main() {
  Vector p1(1.,2.);
```

Member initialization

Other syntax for initialization:

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

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:

```
./pointfunc
p1 has length 2.23607
```

We call such internal functions 'methods'

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:

```
./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:

```
./pointscale
p1 has length 2.23607
p2 has length 4.47214
```

Default constructor

```
Vector p1(1.,2.), p2;
cout << "p1 has length " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 has length " << p2.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 p1(1.,2.), p2;
```

So:

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

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.
- `distance` computes the distance between this point and another: if `p,q` are `Point` objects,

  ```
  p.distance(q)
  ```

  computes the distance.
- `angle` computes the angle of vector (x, y) with the x-axis.

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;
  };
  double x() { return vx; }; // 'accessor'
  double y() { return vy; };
};

int main() {
  Vector p1(1.,2.);
```

Accessor for setting private data

```
void setx( double newx ) { vx = newx; };
void sety( double newy ) { vy = newy; };
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 2

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 3

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;
```

Exercise 4

Write a class `primegenerator` that contains the members of the structure, and the functions `nextprime`, `isprime`. The function `nextprime` does not need the object as argument, because the members are in the object, and therefore global to that function.

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;
}
```

Exercise 5

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 20 million.

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, print out how it is the sum of two primes. If multiple possibilities exist, only print the first one you find.