Objects and classes

Victor Eijkhout, Harika Gurram,
Je'aime Powell, Charley Dey

Fall 2018

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 from running pointstruct in code directory geom:

```
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;
  };



}; // end of class definition

int main() {
  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;
  };


double x() { return vx; };
double y() { return vy; };
```

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



}; // end of class definition

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

Usage:

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

Interface versus implementation

- 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 with two members that sum to one.
You don't want to be able to change just one of them!

```
class SumIsOne {
public:
  float x,y;
  SumIsOne( double xx ) { x = xx; y = 1-x; };
}
int main() {
  SumIsOne pointfive(.5);
  pointfive.y = .6;
}
```

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 initialization

Other syntax for initialization:

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

Even:

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 from running pointinitxy in code directory geom:

```
p1 = 1,2
```

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 from running pointfunc in code directory geom:

```
p1 has length 2.23607
```

We call such internal functions 'methods'.

Data members, even `private`, are global to the 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 from running pointscaleby in code directory geom:

```
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 from running pointscale in code directory geom:

```
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;
```

The problem is with `p2`. How is it created? We need to define two constructors:

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

Exercise 1

```
class Point {
private:
  float x,y;
public:
  Point(float ux,float uy) { x = ux; y = uy; };
  float distance(Point other) {
    float xd = x-other.x, yd = y-other.y;
    return sqrt( xd*xd + yd*yd );
  };
};
```

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;
  };



}; // end of class definition

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

Accessor for setting private data

Class methods:

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

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

```
class LinearFunction {
private:
  Point p1,p2;
public:
  LinearFunction( Point &input_p2 ) {
    p1 = Point(0.,0.);
    p2 = input_p2;
  };
  LinearFunction( Point &input_p1,Point &input_p2 ) {
    p1 = input_p1; p2 = input_p2;
  };
  float evaluate_at( float x ) {
    float slope = (p2.y-p1.y) / (p2.x-p1.x);
    float intercept = p1.y - p1.x * slope;
    return intercept + x*slope;
  };
};
```

Exercise 3

```
class LinearFunction {
private:
  Point p1,p2;
public:
  LinearFunction( Point &input_p2 ) {
    p1 = Point(0.,0.);
    p2 = input_p2;
  };
  LinearFunction( Point &input_p1,Point &input_p2 ) {
    p1 = input_p1; p2 = input_p2;
  };
  float evaluate_at( float x ) {
    float slope = (p2.y-p1.y) / (p2.x-p1.x);
    float intercept = p1.y - p1.x * slope;
    return intercept + x*slope;
  };
};
```

Exercise 4

```
struct primesequence {
  int number_of_primes_found = 0;
  int last_number_tested = 1;
} ;

int nextprime( struct primesequence &sequence ) {
  do {
    sequence.last_number_tested++;
  } while (!isprime(sequence.last_number_tested));
  sequence.number_of_primes_found++;
  return sequence.last_number_tested;
};
```

Exercise 5

```
  primes sequence;
  for (int even=4; even<2000; even+=2) {
    cout << "Testing: " << even << endl;
    bool found = false;
    for ( int p = sequence.firstprime(); p<even ; p=sequence.nextprime() ) {
      int q = even-p;
      if (isprime(q)) {
found = true;
cout << even << "=" << p << "+" << q << endl;
      }
      if (found) break; // stop the q loop
    }
    if (!found) cout << "Stop the presses! Counter example: " << even << endl;
  }
```

More about constructors

Copy constructor

- Several default copy constructors are defined
- They copy an object, recursively.
- You can redefine them as needed.

```
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; };
};
```

Code:

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

Output from running copyscalar in code directory object:

```
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 from running copyvector in code directory object:

```
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.

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;
  }
  cout << "After the nested scope" << endl;
```

Output from running destructor in code directory object:

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

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.

'this'

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

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint;
  };
};
```

This is not often needed. Typical use case: you need to call a function inside a method that needs the object as argument)

```
class someclass;
void somefunction(someclass *c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(this);
};
```