MAT 685: C++ for Mathematicians

Points and classes

John Perry

University of Southern Mississippi

Spring 2017

Outline

1. The `Point` of `class`
2. A `class` of `Point`
3. Summary

Outline

1. The `Point` of `class`
2. A `class` of `Point`
3. Summary

A point in the plane

Two ways to represent a point:

- Cartesian coordinates (x, y)
- Polar coordinates (r, θ)

A point in the plane

Two ways to represent a point:

- Cartesian coordinates (x, y)
- Polar coordinates (r, θ)

C++ does not offer a `Point` type.

How can we work with points? (0/...)

One way:

- variable `Px` is P's x value
- `Py` y value
- `Pr` r value
- `Pt` θ value

How can we work with points? (0/...)

One way:

- variable `Px` is P's x value
- `Py` y value
- `Pr` r value
- `Pt` θ value

Then

- manually track which variables correspond to which point(s)
- pass all four as arguments to functions

How can we work with points? (0/...)

One way:

- variable `Px` is P's x value
- `Py` y value
- `Pr` r value
- `Pt` θ value

Then

- manually track which variables correspond to which point(s)
- pass all four as arguments to functions

This can get confusing.

How can we work with points? (1/...)

Structured programming introduced a way of organizing related data into *fields* of a *structure*:

```
struct Point {
  double x, y, r, theta;
};
```

This defines a new type: `Point`. We can define objects of this type, and access fields using the dot operator:

```
Point a;
a.x = 3.0;
a.y = 2.0;
a.r = sqrt(13);
a.theta = 0.666636;
```

How can we work with points? (1.5/...)

We can also write a function to automatically determine polar from Cartesian:

```
void polar_from_cartesian(Point & P)  {
  P.r = sqrt(P.x*P.x + P.y*P.y);
  P.theta = atan(P.x/P.y);
}
```

Pros and cons of structured approach

Pros

- related data stays together, easier to track

Cons

- all fields must be updated manually, *or*
- must manually pass to function to update other fields, *and*
- structure is modifiable by any fool with a keyboard

Pros and cons of structured approach

Pros

- related data stays together, easier to track

Cons

- all fields must be updated manually, *or*
- must manually pass to function to update other fields, *and*
- structure is modifiable by any fool with a keyboard

```
P.x = 3.0;
P.y = 2.0;
P.r = 1.0; // cuz I sed so
P.theta = 40; // degs rool rads drool!!!
```

How can we work with points? (2/...)

Object-oriented programming beefed up structures into *classes*.
New ideas:

- data hiding
- encapsulation
- inheritance
- overloading
- polymorphism

We briefly describe these ideas

Data hiding

Programmer can restrict use of data points using access specifiers

`public` data can be changed by any fool with a keyboard

```
class Point {
public:
  double x, y, r, theta;
};

Point P;

P.x = 3.0;
P.y = 2.0;
P.r = 1.0; // cuz I sed so
```

Data hiding

Programmer can restrict use of data points using access specifiers

`private` or `protected` data cannot

```
class Point {
protected:
  double x, y, r, theta;
};

Point P;

P.x = 3.0; // compiler error
P.y = 2.0; // compiler error
P.r = 1.0; // compiler error
```

We distinguish `protected` from `private` later — best to use `protected`

Encapsulation

Programmer can link *data* with *algorithms* related to that data

```
class Point {
protected:
  double x, y, r, theta;
public:
  void rotate(double angle);
};

void Point::rotate(double angle) {
  // a miracle occurs here
}

Point P;
P.rotate(3.14159);
```

Inheritance

New classes can inherit data and methods, facilitating code reuse

```
class Colored_Point : public Point {
protected:
  unsigned red, green, blue;
}

Colored_Point CP;
CP.rotate(3.14159); // for free, from Point
```

Overloading

Arithmetic operators can be extended to new types

```
class Abelian_Point : public Point {
  Abelian_Point operator +(Abelian_Point &);
};

Abelian_Point P, Q, R;
R = P + Q;
```

Polymorphism

> *"A word means just what I choose it to mean — neither more nor less."*
>
> — *Humpty Dumpty, Alice Through the Looking Glass*

Functions do different things, depending on their inputs

Polymorphism

"A word means just what I choose it to mean — neither more nor less."

— Humpty Dumpty, Alice Through the Looking Glass

Functions do different things, depending on their inputs

```
long gcd(long, long);
long gcd(long, long, long &, long &);
```

...so you've already seen this in action.

How can we work with points? (2/...)

Object-oriented programming beefed up structures into *classes*.
New ideas:

- data hiding
- encapsulation
- inheritance
- overloading
- ~~polymorphism~~

This chapter focuses on **encapsulation** and **overloading**

Outline

1. The Point of class
2. A class of Point
3. Summary

Interface

What should a `Point` class do?

- initialize
- report values (they're `protected`, after all)
- modify values
  - set
  - rotate
- compare values
  - equality
  - ordering?
- compute values
  - distance
  - midpoint

Interface code

Listing 1: point.hpp (p. 1/2, sans comments)

```
#ifndef __POINT_HPP_
#define __POINT_HPP_

#include <iostream>
using std::ostream;

class Point {

protected:
  double x, y;

public:
  Point(double, double);
  Point(const Point &);

  double get_x() const;
  double get_y() const;
  void set_x(double);
  void set_y(double);
```

Interface code

Listing 2: point.hpp (p. 2/2, sans comments)

```
  double get_radius() const;
  double get_angle() const;
  void set_radius(double);
  void set_angle(double);

  void rotate(double);

  bool operator == (const Point &) const;
  bool operator != (const Point &) const;

};

double distance(const Point &, const Point &);
Point midpoint(const Point &, const Point &);

ostream & operator << (ostream &, const Point &);

#endif
```

Observations on data

- only saving rectangular coordinates
- polar coordinates computed on demand
    - time/space tradeoff
    - memory cheap (these days), computation expensive
    - this implementation a bit unusual

Observations on methods (1/3)

- *Construction*
  - Used to initialize data
  - *Always* named by class

Observations on methods (1/3)

- *Construction*
  - Used to initialize data
  - *Always* named by class
  - Second version is *copy constructor*
    - `Class(const & Class);`
    - often needed for return statements
    - best to implement in general

Observations on methods (1/3)

- *Construction*
  - Used to initialize data
  - *Always* named by class
  - Second version is *copy constructor*
    - `Class(const & Class);`
    - often needed for return statements
    - best to implement in general
  - “default” constructor possible
    - does nothing useful
    - to get it, do not specify a constructor
  - if you provide constructor(s)
    - “default” constructor not created
    - object initialization *must* follow constructor(s)
    - can be a problem for arrays

Observations on methods (2/3)

- "Getters"
  - `get_x()`, `get_y()`, `get_radius()`, `get_angle()`
  - data `protected`; getters used to report values
  - no need to implement for data you want to hide
  - **`const`**: method *does not change object*

Observations on methods (2/3)

- "Getters"
  - `get_x()`, `get_y()`, `get_radius()`, `get_angle()`
  - data `protected`; getters used to report values
  - no need to implement for data you want to hide
  - **`const`**: method *does not change object*
- "Setters"
  - `set_x()`, `set_y()`, `set_radius()`, `set_angle()`
  - data `protected`; setters used to modify values
  - no need to implement for any fool with a keyboard

Observations on methods (3/3)

- Modification
  - `rotate()`

Observations on methods (3/3)

- Modification
    - `rotate()`
- Comparison
    - `operator == ()`, `operator != ()`
    - allows us to compare points in “natural” way
    - **`const`**: method *does not change objects*
    - not required for a class
    - if left undefined, C++ will compare in a “default” way
        - probably not what you want

Other functions

C++ not a “pure” object-oriented language

- not all functions need to be methods

Other functions

C++ not a "pure" object-oriented language

- not all functions need to be methods
- some functions make more sense outside class
  - `distance()` "belongs" to which `Point`?

Other functions

C++ not a "pure" object-oriented language

- not all functions need to be methods
- some functions make more sense outside class
  - `distance()` "belongs" to which `Point`?
- some functions best implemented outside class
  - `operator << ()` and `operator >> ()`
  - *can* be implemented as methods, but often gets ugly

Implementation

Listing 3: point.cpp (1/5)

```
#include "point.hpp"

#include <cmath>
using std::sqrt; using std::cos;
using std::acos; using std::atan2;

Point::Point(double new_x, double new_y) {
  x = new_x; y = new_y;
}

Point::Point(const Point & other) {
  x = other.x; y = other.y;
}

double Point::get_x() const { return x; }

double Point::get_y() const { return y; }
```

Implementation

Listing 4: point.cpp (2/5)

```
void Point::set_x(double new_x) { x = new_x; }

void Point::set_y(double new_y) { y = new_y; }

double Point::get_radius() const {
  return sqrt(x*x + y*y);
}

void Point::set_radius(double r) {
  if (x == 0 and y == 0) { x = r; }
  else {
    double a = get_angle();
    x = r * cos(a);
    y = r * sin(a);
  }
}
```

Implementation

Listing 5: point.cpp (3/5)

```
double Point::get_angle() const {
  double result;
  if (x == 0 and y == 0) result = 0;
  else {
    const double pi = acos(-1);
    result = atan2(y,x);
    if (result < 0) result += 2*pi;
  }
  return result;
}

void Point::set_angle(double theta) {
  double r = get_radius();
  x = r * cos(theta);
  y = r * cos(theta);
}
```

Implementation

Listing 6: point.cpp (4/5)

```
void Point::rotate(double theta) {
  set_angle(get_angle() + theta);
}

bool Point::operator == (const Point & Q) const {
  return (x == Q.x and y == Q.y);
}

bool Point::operator != (const Point & Q) const {
  return (x != Q.x or y != Q.y);
}

double distance(const Point & P, const Point & Q) {
  double dx = P.get_x() - Q.get_x();
  double dy = P.get_y() - Q.get_y();
  return sqrt(dx*dx + dy*dy);
}
```

Implementation

Listing 7: point.cpp (5/5)

```
Point midpoint(const Point & P, const Point & Q) {
  double x = (P.get_x() + Q.get_x()) / 2;
  double y = (P.get_y() + Q.get_y()) / 2;
  return Point(x, y);
}

ostream & operator << (ostream & os, const Point & P)
{
  os << '(' << P.get_x() << ',' << P.get_y() << ')';
  return os;
}
```

Test

Listing 8: test_point.cpp (1/2)

```
#include <iostream>
using std::cout; using std::endl;
#include <cmath>
using std::acos;

#include "point.hpp"

int main() {
  Point X(0,0);
  Point Y(3,4);

  cout << "The point X is " << X
       << " and the point Y is " << Y << endl;
  cout << "Point Y in polar coordinates is ("
       << Y.get_radius() << ','
       << Y.get_angle() << ")\n";
  cout << "The distance between these points is "
       << distance(X, Y) << endl;
```

Test

Listing 9: test_point.cpp (2/2)

```
  cout << "The midpoint between these points is "
       << midpoint(X, Y) << endl;
  const double pi = acos(-1);
  Y.rotate(pi/2);
  cout << "After a 90 degree rotation, Y is "
       << Y << endl;
  Y.set_radius(100);
  cout << "After rescaling, Y is " << Y << endl;
  Point Z(Y);
  cout << "After setting Z to Y, Z is " << Z << endl;
  X = Point(5,3);
  Y = Point(5,-3);
  cout << "Now X is " << X << ", Y is " << Y << endl;
  if (X == Y)
    cout << "They are equal\n";
  if (X != Y)
    cout << "They are not equal\n";
}
```

Compile, execute test

```
$ g++ -c point.cpp
$ g++ -o test_point point.o test_point.cpp
$ ./test_point
The point X is (0,0) and the point Y is (3,4)
Point Y in polar coordinates is (5,0.927295)
The distance between these points is 5
The midpoint between these points is (1.5,2)
After a 90 degree rotation, Y is (-4,-4)
After rescaling, Y is (-70.7107,-70.7107)
After setting Z to Y, Z is (-70.7107,-70.7107)
Now X is (5,3) and Y is (5,-3)
They are not equal
```

Homework

pp. 112–114 #6.1, 6.2, 6.3, 6.6, 6.7

Outline

Summary

- Math stuff
  - None, really
- Programming stuff
  - object-oriented programming
  - encapsulation
  - classes