Class Definition (.h file)

STYLE TIP

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y);     // constructor
  int get_x() const { return x_; }     // inline member function
  int get_y() const { return y_; }     // inline member function
  double Distance(const Point& p) const;      // member function
  void SetLocation(const int x, const int y); // member function

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point

#endif  // POINT_H_
```

const means the object we are calling on, can't be changed

Inline definition ok for simple getters/setters

Declarations

Google C++ naming conventions for data members

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
    - Which one is invoked depends on *how* the object is instantiated
  - A constructor is always invoked when creating a new instance of an object.

- Written with the class name as the method name:

```
Point(const int x, const int y);
```

Zero arg

  - C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors

Created for you

    - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members

Initialization vs. Construction

STYLE TIP

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y, const int z)
    : y_(y), x_(x) {
    z_ = z;
  }
 private:
  int x_, y_, z_;  // data members
};  // class Point3D
```

First, initialization list is applied.

2) set y_ 1) set x_

3) set z_ (garbage)

4) set z_

Next, constructor body is executed.

  - Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (!)
    - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed
  - Initialization preferred to assignment to avoid extra steps
    - Real code should never mix the two styles

Copy Constructors

STYLE TIP

- C++ has the notion of a copy constructor (cctor)
  - Used to create a new object as a copy of an existing object

```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& copyme) {
  x_ = copyme.x_;
  y_ = copyme.y_;
}

void foo() {
  Point x(1, 2);  // invokes the 2-int-arguments constructor
  Point y(x);     // invokes the copy constructor
                  // could also be written as "Point y = x;"
}
```

Reference to object of same type

Use a cctor since we are constructing based on x

Point y didn't exist before, a ctor must be called

  - Initializer lists can also be used in copy constructors (preferred)

When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point x;       // default ctor
Point y(x);    // copy ctor
Point z = y;   // copy ctor
```

  - You pass a non-reference object as a value parameter to a function:

```
void foo(Point x) { ... }

Point y;       // default ctor
foo(y);        // copy ctor
```

  - You return a non-reference object value from a function:

```
Point foo() {
  Point y;     // default ctor
  return y;    // copy ctor
}
```


Overloading the "=" Operator

STYLE TIP

- You can choose to define the "=" operator
  - But there are some rules you should follow:

```
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) {  // (1) always check against this
    x_ = rhs.x_;
    y_ = rhs.y_;
  }
  return *this;        // (2) always return *this from op=
}

Point a;         // default constructor
a = b = c;       // works because = return *this
a = (b = c);     // equiv. to above (= is right-associative)
(a = b) = c;     // "works" because = returns a non-const
                 // reference to *this
```

More important when data members are Dynamic memory

Should be a reference to *this to allow chaining

Explicit equivalent:

```
a.operator=(b.operator=(c));
```


Destructors

- C++ has the notion of a destructor (dtor)
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: "*Resource Acquisition Is Initialization*" (RAII)

tilde

No parameters

```
Point::~Point() {    // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

When a destructor is invoked:
1. run destructor body
2. Call destructor of any data members