

Classes and Objects

member variables

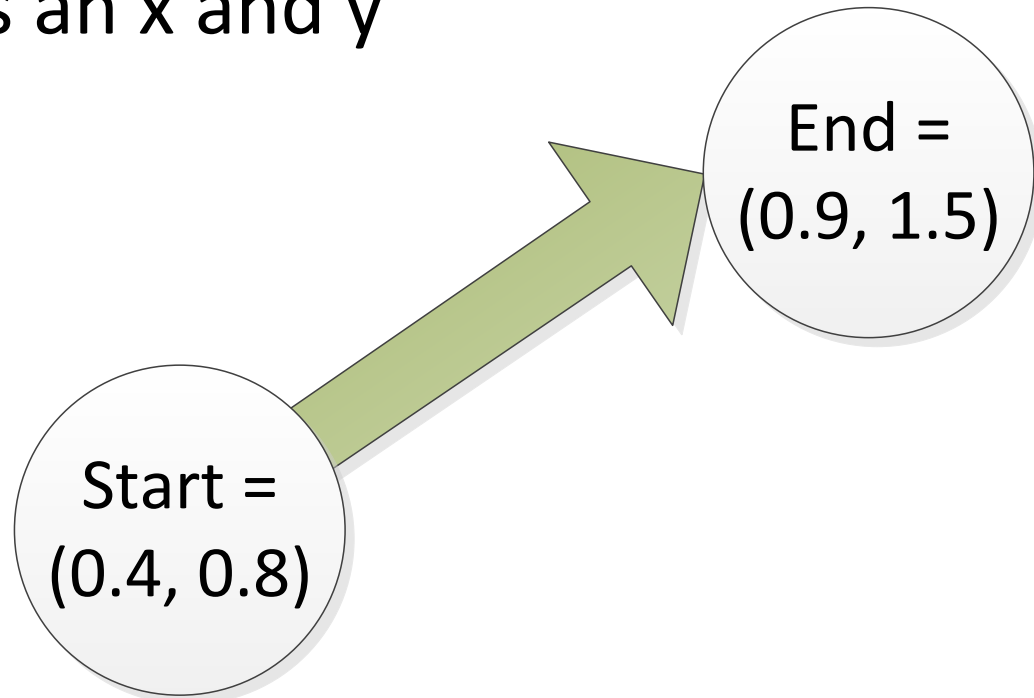
member functions

constructors

access specifiers

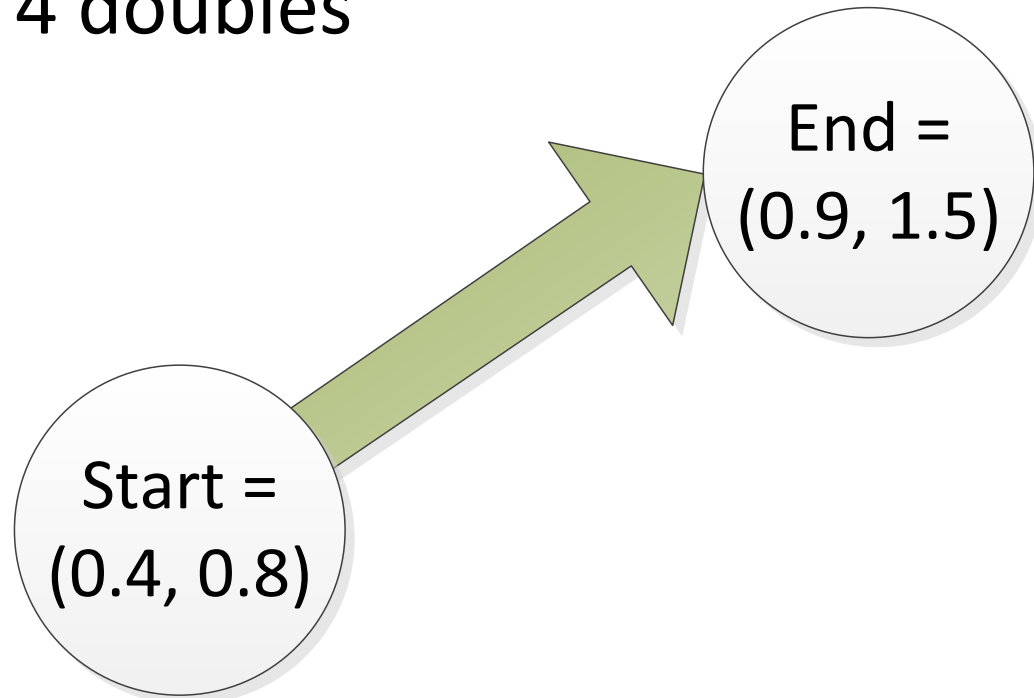
Representing a (Geometric) Vector

- In the context of geometry, a vector consists of 2 points: a start and a finish
- Each point itself has an x and y coordinate

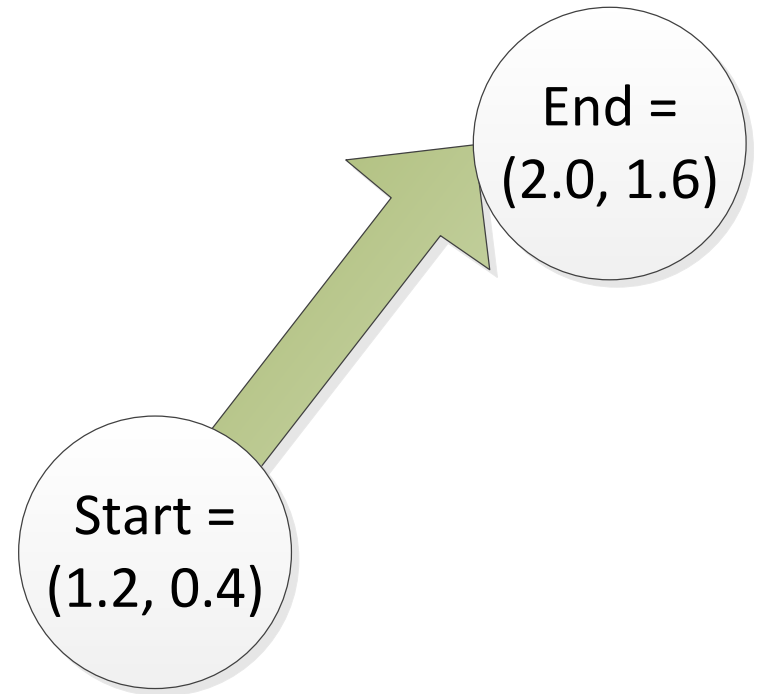


Representing a (Geometric) Vector

- Our representation so far? Use 4 doubles (startx, starty, endx, endy)
- We need to pass all 4 doubles to functions



```
int main() {  
    double xStart = 1.2;  
    double xEnd = 2.0;  
    double yStart = 0.4;  
    double yEnd = 1.6;  
}
```



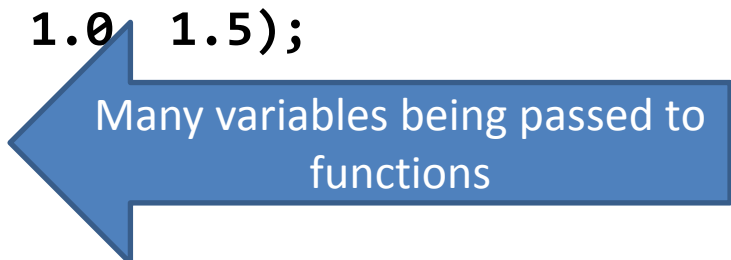
```
void printVector(double x0, double x1, double y0, double y1) {  
    cout << "(" << x0 << "," << y0 << ") -> ("  
        << x1 << "," << y1 << ")" << endl;  
}
```

```
int main() {  
    double xStart = 1.2;  
    double xEnd = 2.0;  
    double yStart = 0.4;  
    double yEnd = 1.6;  
    printVector(xStart, xEnd, yStart, yEnd);  
    // (1.2,2.0) -> (0.4,1.6)  
}
```

```
void offsetVector(double &x0, double &x1, double &y0, double &y1,  
                 double offsetX, double offsetY) {  
    x0 += offsetX;  
    x1 += offsetX;  
    y0 += offsetY;  
    y1 += offsetY;  
}
```

```
void printVector(double x0, double x1, double y0, double y1) {  
    cout << "(" << x0 << "," << y0 << ") -> ("  
        << x1 << "," << y1 << ")" << endl;  
}
```

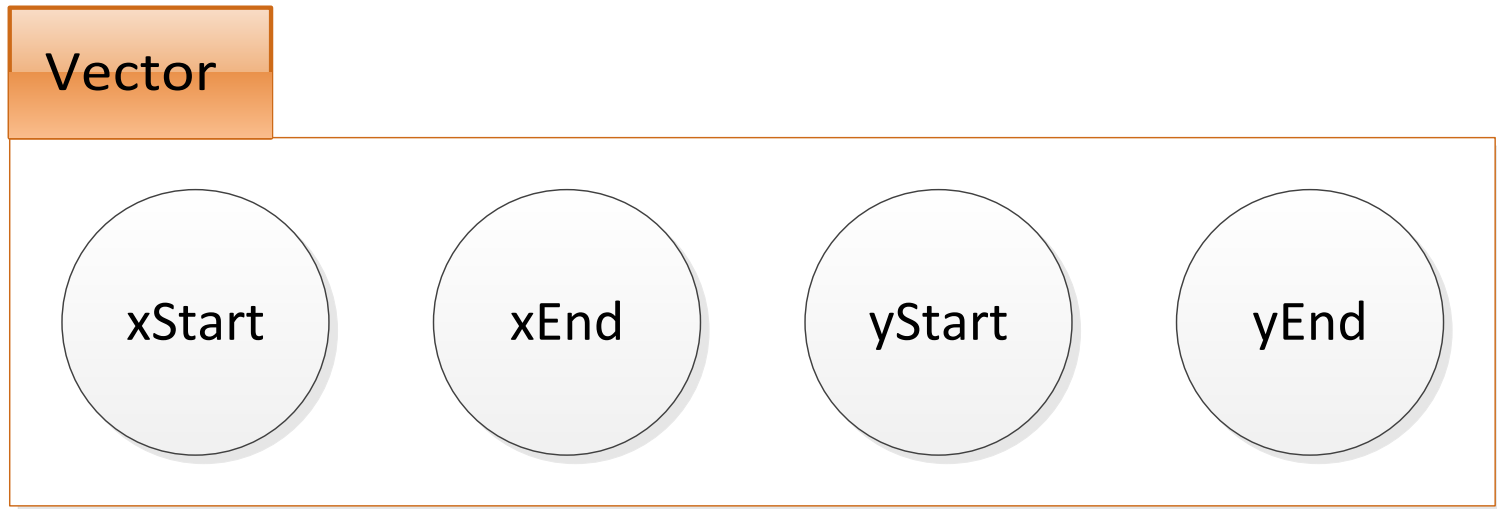
```
int main() {  
    double xStart = 1.2;  
    double xEnd = 2.0;  
    double yStart = 0.4;  
    double yEnd = 1.6;  
    offsetVector(xStart, xEnd, yStart, yEnd, 1.0, 1.5);  
    printVector(xStart, xEnd, yStart, yEnd);  
    // (2.2,1.9) -> (3.8,4.3)  
}
```



Many variables being passed to functions

class

- A user-defined datatype which groups together related pieces of information



class definition syntax



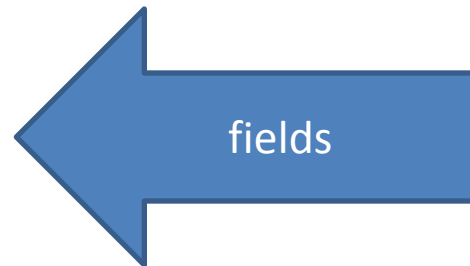
name

```
class Vector {  
public:  
    double xStart;  
    double xEnd;  
    double yStart;  
    double yEnd;  
};
```

- This indicates that the new datatype we're defining is called Vector

class definition syntax

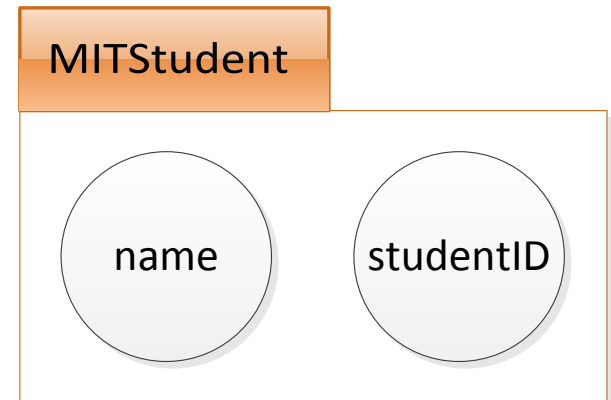
```
class Vector {  
public:  
    double xStart;  
    double xEnd;  
    double yStart;  
    double yEnd;  
};
```



- **Fields** indicate what related pieces of information our datatype consists of
 - Another word for field is **members**

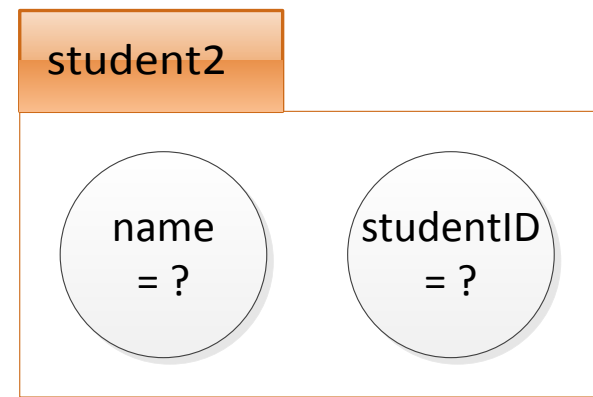
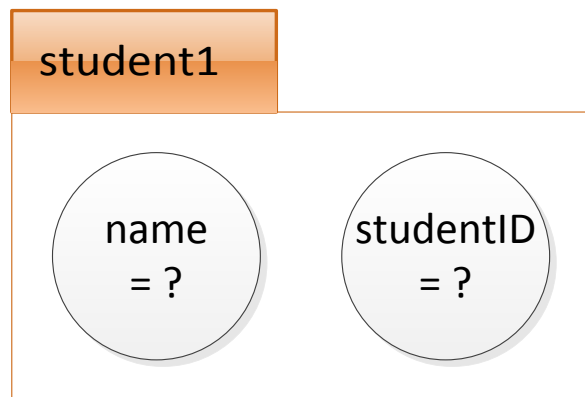
Fields can have different types

```
class MITStudent {  
public:  
    char *name;  
    int studentID;  
};
```



Instances

- An instance is an occurrence of a class. Different instances can have their own set of values in their fields.
- If you wanted to represent 2 different students (who can have different names and IDs), you would use 2 instances of MITStudent



Declaring an Instance

- Defines 2 instances of MITStudent: one called student1, the other called student2

```
class MITStudent {  
public:  
    char *name;  
    int studentID;  
};  
  
int main() {  
    MITStudent student1;  
    MITStudent student2;  
}
```

student1

name
= ?

studentID
= ?

student2

name
= ?

studentID
= ?

Accessing Fields

- To access fields of instances, use `variable.fieldName`

```
class MITStudent {  
public:  
    char *name;  
    int studentID;  
};  
  
int main() {  
    MITStudent student1;  
    MITStudent student2;  
    student1.name = "Geza";  
}
```

student1

name
= "Geza"

studentID
= ?

student2

name
= ?

studentID
= ?

Accessing Fields

- To access fields of instances, use `variable.fieldName`

```
class MITStudent {  
public:  
    char *name;  
    int studentID;  
};  
  
int main() {  
    MITStudent student1;  
    MITStudent student2;  
    student1.name = "Geza";  
    student1.studentID = 123456789;  
}
```

student1

name
= "Geza"

studentID
= 123456789

student2

name
= ?

studentID
= ?

Accessing Fields

- To access fields of instances, use `variable.fieldName`

```
class MITStudent {
public:
    char *name;
    int studentID;
};

int main() {
    MITStudent student1;
    MITStudent student2;
    student1.name = "Geza";
    student1.studentID = 123456789;
    student2.name = "Jesse";
    student2.studentID = 987654321;
}
```

student1

name
= "Geza"

studentID
= 123456789

student2

name
= "Jesse"

studentID
= 987654321

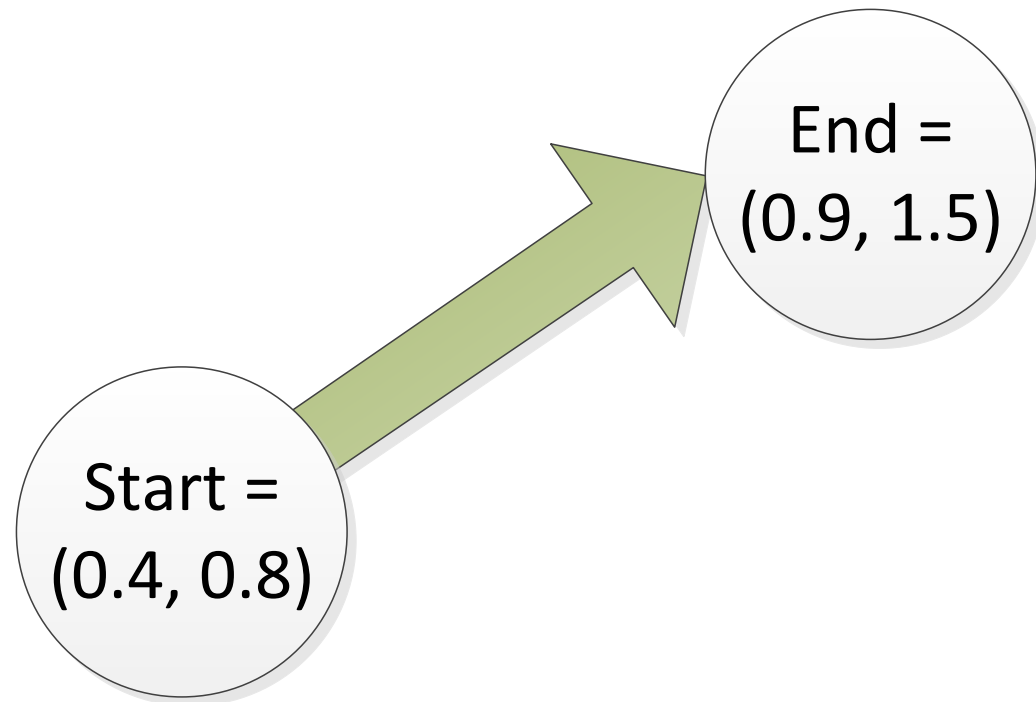
Accessing Fields

- To access fields of instances, use `variable.fieldName`

```
class MITStudent {  
public:  
    char *name;  
    int studentID;  
};
```

```
int main() {  
    MITStudent student1;  
    MITStudent student2;  
    student1.name = "Geza";  
    student1.studentID = 123456789;  
    student2.name = "Jesse";  
    student2.studentID = 987654321;  
    cout << "student1 name is" << student1.name << endl;  
    cout << "student1 id is" << student1.studentID << endl;  
    cout << "student2 name is" << student2.name << endl;  
    cout << "student2 id is" << student2.studentID << endl;
```


- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish



- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish

```
class Vector {  
public:  
    double xStart;  
    double xEnd;  
    double yStart;  
    double yEnd;  
};
```

Vector

xStart

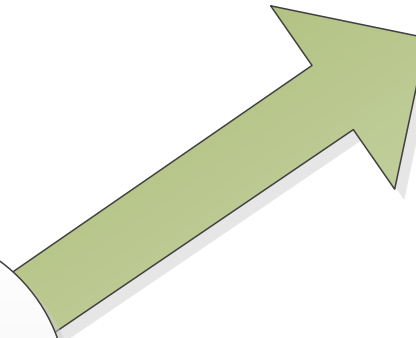
xEnd

yStart

yEnd

Start =
(0.4, 0.8)

End =
(0.9, 1.5)



- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish

```
class Vector {  
public:  
    double xStart;  
    double xEnd;  
    double yStart;  
    double yEnd;  
};
```

Doesn't show that coordinates
can be grouped into points

Vector

xStart

xEnd

yStart

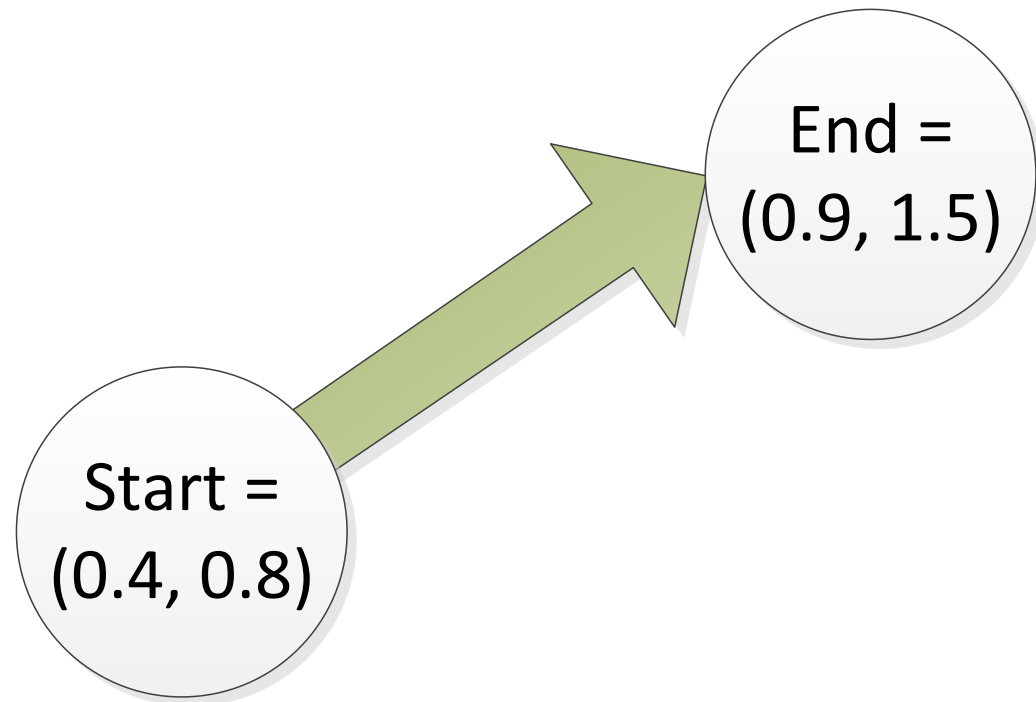
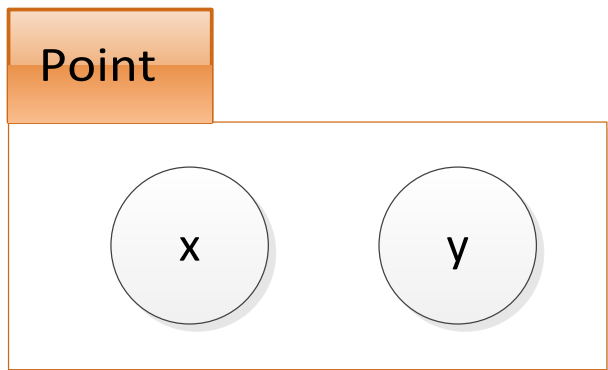
yEnd

Start =
(0.4, 0.8)

End =
(0.9, 1.5)

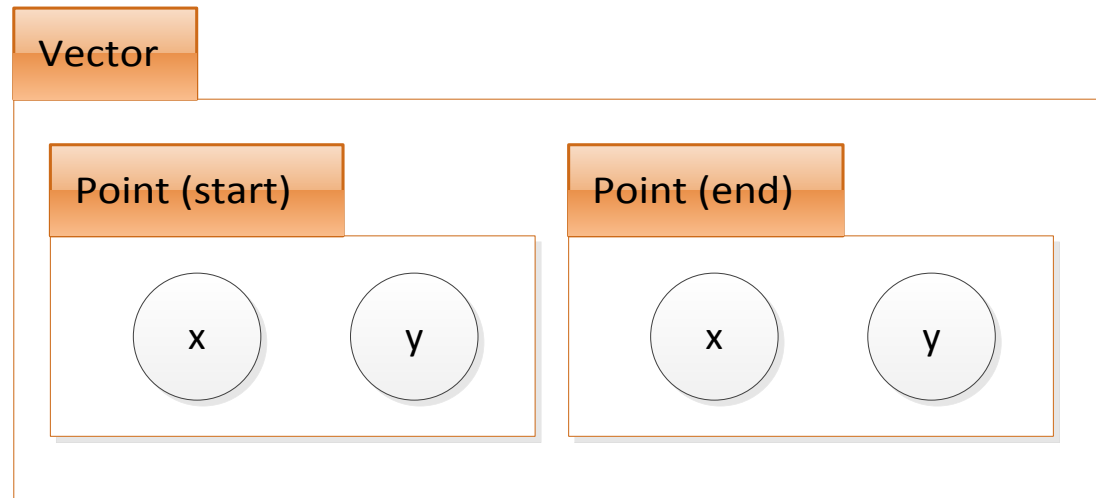
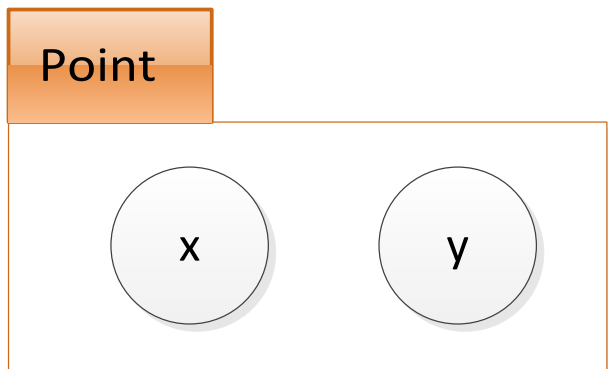
```
class Point {  
public:  
    double x;  
    double y;  
};
```

- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish



```
class Point {  
public:  
    double x;  
    double y;  
};
```

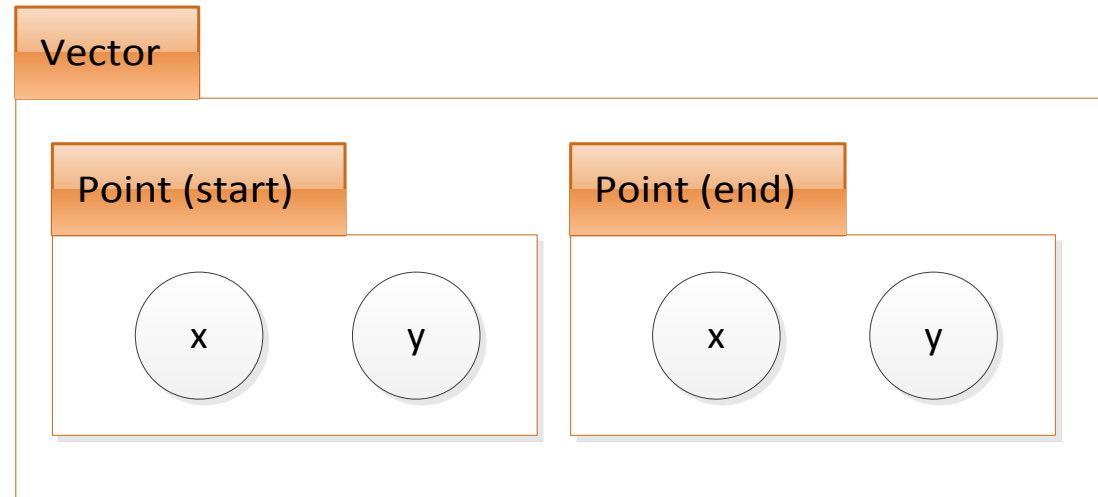
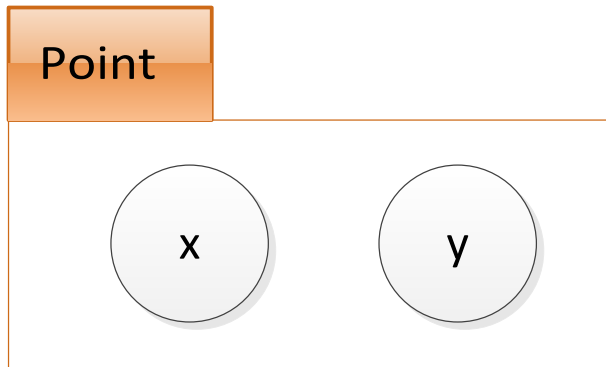
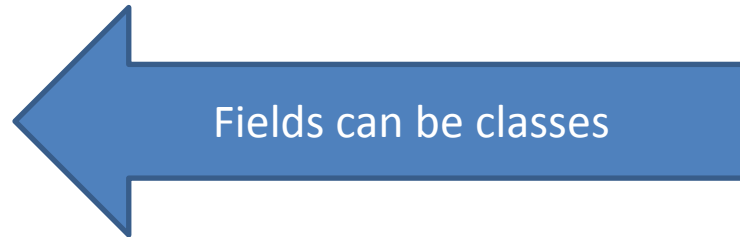
- A point consists of an x and y coordinate
- **A vector consists of 2 points: a start and a finish**



- A point consists of an x and y coordinate
- **A vector consists of 2 points: a start and a finish**

```
class Point {  
public:  
    double x;  
    double y;  
};
```

```
class Vector {  
public:  
    Point start;  
    Point end;  
};
```



```
class Point {  
public:  
    double x, y;  
};  
  
class Vector {  
public:  
    Point start, end;  
};  
  
int main() {  
    Vector vec1;  
}
```

vec1 (instance of Vector)

start (instance of Point)

x=?

y=?

end (instance of Point)

x=?

y=?

```
class Point {  
public:  
    double x, y;  
};  
  
class Vector {  
public:  
    Point start, end;  
};  
  
int main() {  
    Vector vec1;  
    vec1.start.x = 3.0;  
}
```

vec1 (instance of Vector)

start (instance of Point)

x=3

y=?

end (instance of Point)

x=?

y=?


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

class Vector {
public:
    Point start, end;
};
```

```
int main() {
    Vector vec1;
    vec1.start.x = 3.0;
    vec1.start.y = 4.0;
    vec1.end.x = 5.0;
    vec1.end.y = 6.0;
}
```

vec1 (instance of Vector)

start (instance of Point)

x=3

y=4

end (instance of Point)

x=5

y=6

```

class Point {
public:
    double x, y;
};

class Vector {
public:
    Point start, end;
};

```

```

int main() {
    Vector vec1;
    vec1.start.x = 3.0;
    vec1.start.y = 4.0;
    vec1.end.x = 5.0;
    vec1.end.y = 6.0;
    Vector vec2;
}

```

vec1 (instance of Vector)

start (instance of Point)

x=3

y=4

end (instance of Point)

x=5

y=6

vec2 (instance of Vector)

start (instance of Point)

x=?

y=?

end (instance of Point)

x=?

y=?

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

```
class Vector {
public:
    Point start, end;
};
```

```
int main() {
    Vector vec1;
    vec1.start.x = 3.0;
    vec1.start.y = 4.0;
    vec1.end.x = 5.0;
    vec1.end.y = 6.0;
    Vector vec2;
    vec2.start = vec1.start;
}
```

vec1 (instance of Vector)

start (instance of Point)

x=3

y=4

end (instance of Point)

x=5

y=6

vec2 (instance of Vector)

start (instance of Point)

x=3

y=4

end (instance of Point)

x=?

y=?

- Assigning one instance to another copies all fields

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

```
class Vector {
public:
    Point start, end;
};
```

```
int main() {
    Vector vec1;
    vec1.start.x = 3.0;
    vec1.start.y = 4.0;
    vec1.end.x = 5.0;
    vec1.end.y = 6.0;
    Vector vec2;
    vec2.start = vec1.start;
    vec2.start.x = 7.0;
}
```

vec1 (instance of Vector)

start (instance of Point)

x=3

y=4

end (instance of Point)

x=5

y=6

vec2 (instance of Vector)

start (instance of Point)

x=7

y=4

end (instance of Point)

x=?

y=?

- Assigning one instance to another copies all fields

Passing classes to functions

- Passing by value passes a copy of the class instance to the function; changes aren't preserved

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

```
void offsetPoint(Point p, double x, double y) { // does nothing  
    p.x += x;  
    p.y += y;  
}
```

```
int main() {  
    Point p;  
    p.x = 3.0;  
    p.y = 4.0;  
    offsetPoint(p, 1.0, 2.0); // does nothing  
    cout << "(" << p.x << "," << p.y << ")"; // (3.0,4.0)  
}
```

Passing classes to functions

- When a class instance is passed by reference, changes are reflected in the original

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

```
void offsetPoint(Point &p, double x, double y) { // works  
    p.x += x;  
    p.y += y;  
}
```

Passed by
reference

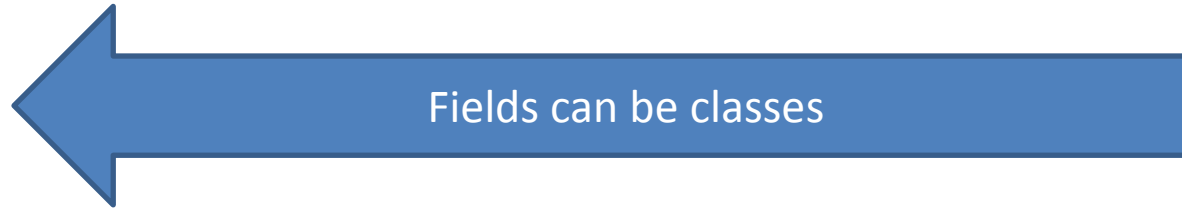
```
int main() {  
    Point p;  
    p.x = 3.0;  
    p.y = 4.0;  
    offsetPoint(p, 1.0, 2.0); // works  
    cout << "(" << p.x << ", " << p.y << ")"; // (4.0,6.0)  
}
```

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



Point class, with fields x and y

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




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

```
int main() {  
    Vector vec;  
}
```



vec is an instance of Vector

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

```
int main() {  
    Vector vec;  
    vec.start.x = 1.2;  
}
```



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

```
int main() {  
    Vector vec;  
    vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;  
}
```

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

class Vector {
    public: Point start, end;
};

void printVector(Vector v) {
    cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
", " << v.end.y << ")" << endl;
}

int main() {
    Vector vec;
    vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
    printVector(vec); // (1.2,0.4) -> (2.0,1.6)
}
```

classes can be passed
to functions

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

class Vector {
    public: Point start, end;
};
```

Can pass to value if you don't
need to modify the class

```
void printVector(Vector v) {
    cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
    "," << v.end.y << ")" << endl;
}

int main() {
    Vector vec;
    vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
    printVector(vec); // (1.2,0.4) -> (2.0,1.6)
}
```

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

Pass classes by reference if they need to be modified

```
void offsetVector(Vector &v, double offsetX, double offsetY) {
    v.start.x += offsetX;
    v.end.x += offsetX;
    v.start.y += offsetY;
    v.end.y += offsetY;
}
void printVector(Vector v) {
    cout << "(" << v.start.x << "," << v.start.y << ") -> (" << v.end.x <<
", " << v.end.y << ")" << endl;
}

int main() {
    Vector vec;
    vec.start.x = 1.2; vec.end.x = 2.0; vec.start.y = 0.4; vec.end.y = 1.6;
    offsetVector(vec, 1.0, 1.5);
    printVector(vec); // (2.2,1.9) -> (3.8,4.3)
}
```

- Observe how some functions are closely associated with a particular class

```
void offsetVector(Vector &v, double offsetX, double offsetY);  
void printVector(Vector v);
```

```
int main() {  
    Vector vec;  
    vec.start.x = 1.2; vec.end.x = 2.0;  
    vec.start.y = 0.4; vec.end.y = 1.6;  
    offsetVector(vec, 1.0, 1.5);  
    printVector(vec);  
}
```

- Observe how some functions are closely associated with a particular class
- **Methods:** functions which are part of a class

```
Vector vec;  
vec.start.x = 1.2; vec.end.x = 2.0;  
vec.start.y = 0.4; vec.end.y = 1.6;  
vec.print();
```



Method name

- Observe how some functions are closely associated with a particular class
- **Methods:** functions which are part of a class
 - Implicitly pass the current instance

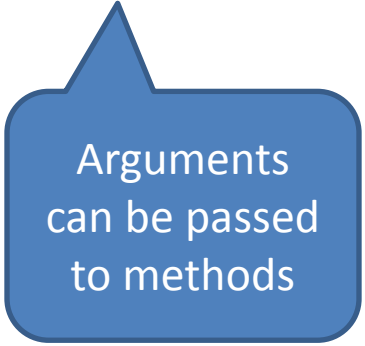
```
Vector vec;  
vec.start.x = 1.2; vec.end.x = 2.0;  
vec.start.y = 0.4; vec.end.y = 1.6;  
vec.print();
```



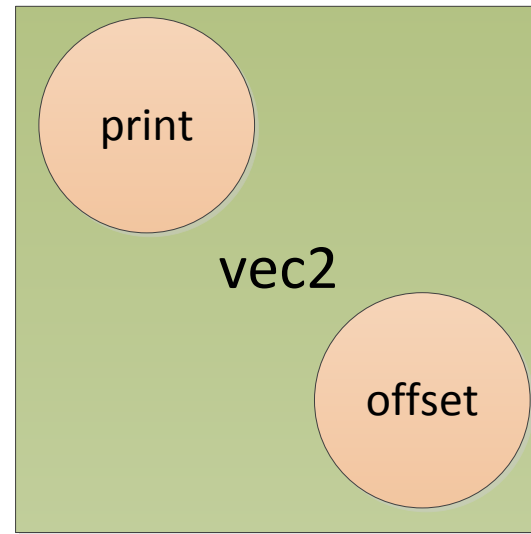
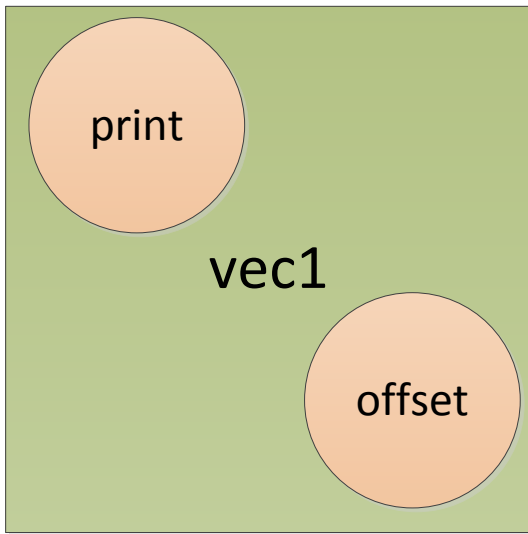
Object
instance

- Observe how some functions are closely associated with a particular class
- **Methods:** functions which are part of a class
 - Implicitly pass the current instance

```
Vector vec;  
vec.start.x = 1.2; vec.end.x = 2.0;  
vec.start.y = 0.4; vec.end.y = 1.6;  
vec.print();  
vec.offset(1.0, 1.5);
```

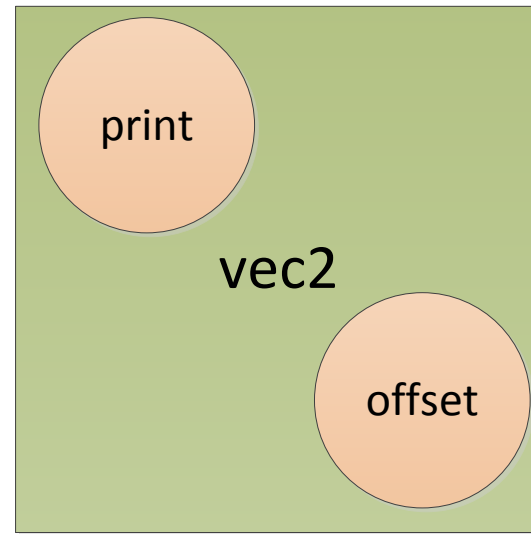
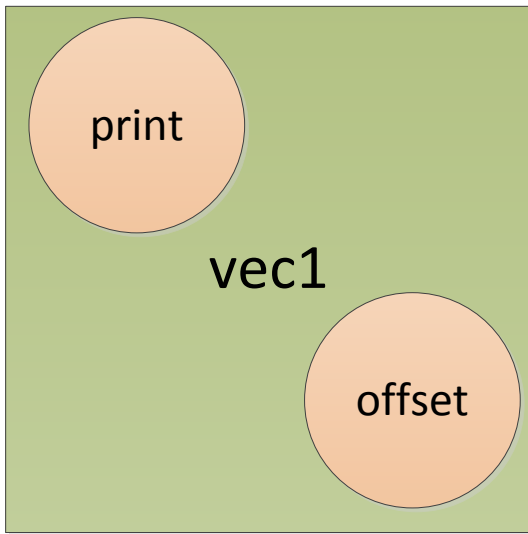


Arguments
can be passed
to methods



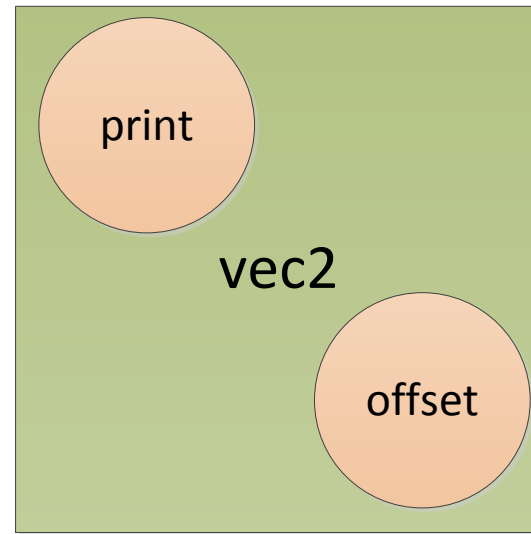
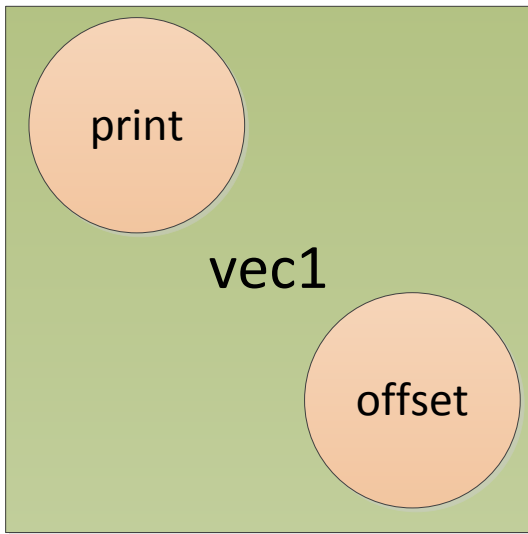
```
Vector vec1;  
Vector vec2;  
// initialize vec1 and vec2  
vec1.print();
```

- Analogy: Methods are “buttons” on each box (instance), which do things when pressed



```
Vector vec1;  
Vector vec2;  
// initialize vec1 and vec2  
vec1.print();
```

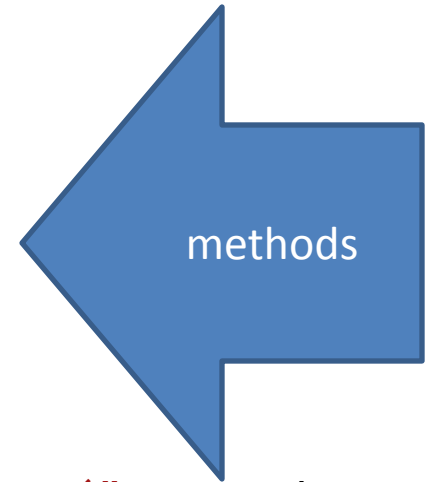
Which box's
button was
pressed?



```
Vector vec1;  
Vector vec2;  
// initialize vec1 and vec2  
vec1.print();
```

Which button
was pressed?

```
class Vector {  
public:  
    Point start;  
    Point end;  
  
    void offset(double offsetX, double offsetY) {  
        start.x += offsetX;  
        end.x += offsetX;  
        start.y += offsetY;  
        end.y += offsetY;  
    }  
    void print() {  
        cout << "(" << start.x << "," << start.y << ") -> (" << end.x <<  
        "," << end.y << ")" << endl;  
    }  
};
```



```
class Vector {  
public:  
    Point start;  
    Point end;  
  
    void offset(double offsetX, double offsetY) {  
        start.x += offsetX;  
        end.x += offsetX;  
        start.y += offsetY;  
        end.y += offsetY;  
    }  
    void print() {  
        cout << "(" << start.x << ", " << start.y << ") -> (" << end.x <<  
        ", " << end.y << ")" << endl;  
    }  
};
```



Fields can be accessed in a method

```
class Vector {  
public:  
    Point start, end;
```

```
    void offset(double offsetX, double offsetY) {  
        start.offset(offsetX, offsetY);  
        end.offset(offsetX, offsetY);  
    }
```

```
    void print() {  
        start.print();  
        cout << " -> ";  
        end.print();  
        cout << endl;  
    }
```

```
};
```



methods of fields can be called

```
class Point {  
public:
```

```
    double x, y;
```

```
    void offset(double offsetX, double offsetY) {  
        x += offsetX; y += offsetY;  
    }
```

```
    void print() {  
        cout << "(" << x << ", " << y << ")";  
    }
```

```
};
```


Implementing Methods Separately

- Recall that function prototypes allowed us to declare that functions will be implemented later
- This can be done analogously for class methods

```
// vector.h - header file
class Point {
public:
    double x, y;
    void offset(double offsetX, double offsetY);
    void print();
};

class Vector {
public:
    Point start, end;
    void offset(double offsetX, double offsetY);
    void print();
};
```

```
#include "vector.h"
// vector.cpp - method implementation
void Point::offset(double offsetX, double offsetY) {
    x += offsetX; y += offsetY;
}
void Point::print() {
    cout << "(" << x << ", " << y << ")";
}
void Vector::offset(double offsetX, double offsetY) {
    start.offset(offsetX, offsetY);
    end.offset(offsetX, offsetY);
}
void Vector::print() {
    start.print();
    cout << " -> ";
    end.print();
    cout << endl;
}
```



:: indicates which class' method is being implemented

- Manually initializing your fields can get tedious
- Can we initialize them when we create an instance?

```
Vector vec;  
vec.start.x = 0.0;  
vec.start.y = 0.0;  
vec.end.x = 0.0;  
vec.end.y = 0.0;
```

```
Point p;  
p.x = 0.0;  
p.y = 0.0;
```

Constructors

- Method that is called when an instance is created

```
class Point {  
public:  
    double x, y;  
    Point() {  
        x = 0.0; y = 0.0; cout << "Point instance created" << endl;  
    }  
};  
  
int main() {  
    Point p; // Point instance created  
    // p.x is 0.0, p.y is 0.0  
}
```

Constructors

- Can accept parameters

```
class Point {  
public:  
    double x, y;  
    Point(double nx, double ny) {  
        x = nx; y = ny; cout << "2-parameter constructor" << endl;  
    }  
};  
  
int main() {  
    Point p(2.0, 3.0); // 2-parameter constructor  
    // p.x is 2.0, p.y is 3.0  
}
```

Constructors

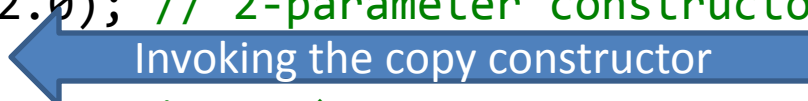
- Can have multiple constructors

```
class Point {
public:
    double x, y;
    Point() {
        x = 0.0; y = 0.0; cout << "default constructor" << endl;
    }
    Point(double nx, double ny) {
        x = nx; y = ny; cout << "2-parameter constructor" << endl;
    }
};

int main() {
    Point p; // default constructor
    // p.x is 0.0, p.y is 0.0)
    Point q(2.0, 3.0); // 2-parameter constructor
    // q.x is 2.0, q.y is 3.0)
}
```

- Recall that assigning one class instance to another copies all fields (default **copy constructor**)

```
class Point {
public:
    double x, y;
    Point() {
        x = 0.0; y = 0.0; cout << "default constructor" << endl;
    }
    Point(double nx, double ny) {
        x = nx; y = ny; cout << "2-parameter constructor" << endl;
    }
};
```

```
int main() {
    Point q(1.0, 2.0); // 2-parameter constructor
    Point r = q;  Invoking the copy constructor
    // r.x is 1.0, r.y is 2.0
}
```


- You can define your own copy constructor

```
class Point {  
public:  
    double x, y;  
    Point(double nx, double ny) {  
        x = nx; y = ny; cout << "2-parameter constructor" << endl;  
    }  
    Point(Point &o) {  
        x = o.x; y = o.y; cout << "custom copy constructor" << endl;  
    }  
};  
  
int main() {  
    Point q(1.0, 2.0); // 2-parameter constructor  
    Point r = q; // custom copy constructor  
    // r.x is 1, r.y is 2  
}
```


- Why make a copy constructor? Assigning all fields (default copy constructor) may not be what you want

```
class MITStudent {  
public:  
    int studentID;  
    char *name;  
    MITStudent() {  
        studentID = 0;  
        name = "";  
    }  
};
```

```
int main() {  
    MITStudent student1;  
    student1.studentID = 98;  
    char n[] = "foo";  
    student1.name = n;  
    MITStudent student2 = student1;  
    student2.name[0] = 'b';  
    cout << student1.name; // boo  
}
```



By changing student 2's name, we changed student 1's name as well

- Why make a copy constructor? Assigning all fields (default copy constructor) may not be what you want

```
class MITStudent {
public:
    int studentID;
    char *name;
    MITStudent() {
        studentID = 0;
        name = "";
    }
    MITStudent(MITStudent &o) {
        studentID = o.studentID;
        name = strdup(o.name);
    }
};
```

```
int main() {
    MITStudent student1;
    student1.studentID = 98;
    char n[] = "foo";
    student1.name = n;
    MITStudent student2 = student1;
    student2.name[0] = 'b';
    cout << student1.name; // foo
}
```

Changing student 2's name doesn't effect student 1's name

Access Modifiers

- Define where your fields/methods can be accessed from



Access Modifier

```
class Point {  
    public:  
        double x, y;  
  
    Point(double nx, double ny) {  
        x = nx; y = ny;  
    }  
};
```

Access Modifiers

- public: can be accessed from anywhere

```
class Point {  
public:  
    double x, y;  
  
    Point(double nx, double ny) {  
        x = nx; y = ny;  
    }  
};
```

```
int main() {  
    Point p(2.0,3.0);  
    p.x = 5.0; // allowed  
}
```

Access Modifiers

- private: can only be accessed within the class

```
class Point {  
private:  
    double x, y;  
  
public:  
    Point(double nx, double ny) {  
        x = nx; y = ny;  
    }  
};  
  
int main() {  
    Point p(2.0,3.0);  
    p.x = 5.0; // not allowed  
}
```

Access Modifiers

- Use getters to allow read-only access to private fields

```
class Point {  
private:  
    double x, y;  
  
public:  
    Point(double nx, double ny) {  
        x = nx; y = ny;  
    }  
    double getX() { return x; }  
    double getY() { return y; }  
};  
  
int main() {  
    Point p(2.0,3.0);  
    cout << p.getX() << endl; // allowed  
}
```

Default Access Modifiers

- class: private by default

```
class Point {  
    double x, y;  
};
```



```
class Point {  
private:  
    double x, y;  
};
```

Structs

- Structs are a carry-over from the C; in C++, classes are generally used
- In C++, they're essentially the same as classes, except structs' default access modifier is public

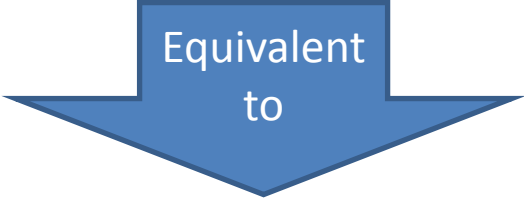
```
class Point {  
    public:  
        double x;  
        double y;  
};
```

```
struct Point {  
    double x;  
    double y;  
};
```


Default Access Modifiers

- struct: public by default
- class: private by default

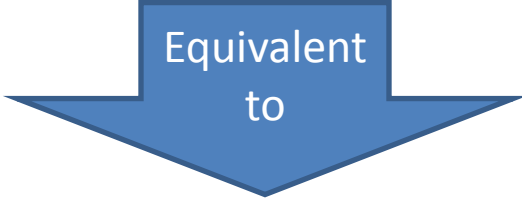
```
struct Point {  
    double x, y;  
};
```



Equivalent
to

```
struct Point {  
public:  
    double x, y;  
};
```

```
class Point {  
    double x, y;  
};
```



Equivalent
to

```
class Point {  
private:  
    double x, y;  
};
```