CS 180 - C++: References + Pointers

Announcements

- HW1 due Wednesday
- Program 1 due next Friday -
  U check point next Tuesday
- Lab on Friday this week
- Tutoring hours are posted on department webpage
Last time:

- input & output: iostream, fstream
- classes & member data/functions
  Point class
  C++ forces private data
Simple Point Class

class Point {
    private:
        double _x;
        double _y;
    
    public:
        Point() : _x(0), _y(0) {} // constructor
        double getX() const {
            return _x;
        } // accessor
        void setX(double val) {
            _x = val;
        } // mutator
        double getY() const {
            return _y;
        } // accessor
        void setY(double val) {
            _y = val;
        } // mutator

}; // end of Point class (semicolon is required)
Robust Point Class:

class Point {
    private:
        double _x;
        double _y;
    
    public:
        Point(double initialX=0.0, double initialY=0.0) : _x(initialX), _y(initialY) {} 

        double getX() const { return _x; } // same as simple Point class
        void setX(double val) { _x = val; } // same as simple Point class
        double getY() const { return _y; } // same as simple Point class
        void setY(double val) { _y = val; } // same as simple Point class

        void scale(double factor) { 
            _x *= factor;                  // _x = _x * factor
            _y *= factor;
        }

        ;
        ;
Robust Point class cont:

double distance(Point other) const {
    double dx = .x - other.x;
    double dy = .y - other.y;
    return sqrt(dx * dx + dy * dy);
} // sqrt imported from cmath library

void normalize() {
    double mag = distance(Point()); // measure distance to the origin
    if (mag > 0)
      scale(1/mag);
}

Point operator+(Point other) const {
    return Point(.x + other.x, .y + other.y);
}

Point operator*(double factor) const {
    return Point(.x * factor, .y * factor);
}

double operator*(Point other) const {
    return .x * other.x + .y * other.y;
}; // end of Point class (semicolon is required)
Things to note:

1) \(-x + \text{other. } -x \leq \text{allowed if insider the class} \) (even though \(-x\) is private)
2) using \text{operator+}, will be \(x+y\)
3) two versions of \(*\)
   one for factors versus one for points
   \((1,1) * 5 = (5, 5)\)
   \((1,1) * (3, 2) = 5\)

another issue: \(5 * (1,1)\)
Additional functions (Not in class)

```cpp
// Free-standing operator definitions, outside the formal Point class definition
Point operator*(double factor, Point p) {
    return p * factor; // invoke existing form with Point as left operand
}

ostream& operator<<(ostream& out, Point p) {
    out << "(" << p.getX() << ", " << p.getY() << ")"; // display using form <x,y>
    return out;
}

Why outside of class?

C++ does not allow right operator to be instance of an object
```
Inheritance - a good way to be lazy

What is it?

Allowing code reuse by declaring sub-class.

Child class "inherits" square shape
all data & functions but additional ones can be added.
Example: Square class

class Square : public Rectangle {
    public:
        Square(double size=10, Point center=Point( )) : 
            Rectangle(size, size, center)  // parent constructor
        { }

    void setHeight(double h) { setSize(h); }
    void setWidth(double w) { setSize(w); }

    void setSize(double size) {
        Rectangle::setWidth(size);  // make sure to invoke PARENT version
        Rectangle::setHeight(size);  // make sure to invoke PARENT version
    }

    double getSize() const { return getWidth(); }
}; // end of Square
Other issues:

A new type of data:
- We have seen public & private.
  Public is inherited and private is not.

But what about data which should be private, but also should be inherited?

Ex: public:
- int height;
- int width;

protected:
- int height;
- int width;
Objects & Memory Management

In Python, variables were pointers to data.

\[ a = \text{Point}(x=3.2, y=-5.8) \]

\[ b = a \]

\[ b = \text{Point}(x=3, y=4) \]

\[ c = a \]

\[ c = c + b \]
C++: A more versatile setup

C++ allows 3 different models for storing & passing information.

1. Value
2. Reference
3. Pointer

(Remember that strange & a few slides ago?)
Value Variables

When a variable is created, a precise amount of memory is set aside:

Point a:
Point b (5, 7):

\[ b = a \cdot \]

<table>
<thead>
<tr>
<th>a : Point</th>
<th>b : Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0.0</td>
<td>x = 5.0</td>
</tr>
<tr>
<td>y = 0.0</td>
<td>y = 7.0</td>
</tr>
</tbody>
</table>

This is more efficient, both for space and speed.
Now suppose we set \( a = b \):

<table>
<thead>
<tr>
<th>a : Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 5.0</td>
</tr>
<tr>
<td>y = 7.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b : Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 5.0</td>
</tr>
<tr>
<td>y = 7.0</td>
</tr>
</tbody>
</table>

They stay separate!

Different than Python:
Functions: Passing by Value

```cpp
bool isOrigin(Point pt) {
    return pt.getX() == 0 && pt.getY() == 0;
}
```

pt. `set X = 5;`

wouldn't change my Point

When someone calls `isOrigin(myPoint)` later, the value `pt` in the function is initialized as though a new variable was created:

```cpp
Point pt(myPoint);
```

So changes in function to `pt` don't affect `myPoint`!
Reference Variables

Syntax: Point& c(a); // reference variable

- c is created as an alias for a
- More like Python model, but can't be changed later

Ex: c = b;
   Will not rebind c to point to b, but will change the value of c (and a).
Passing by reference:
Reference variables aren't usually needed in main program.
Instead, they are primarily used for passing to functions.

Ex:
```cpp
bool isOrigin(Point& pt) {
    return pt.getX() == 0 && pt.getY() == 0;
}
```

Instead of making a local copy of input, makes a reference here, changes to `pt` persist outside.
Passing by reference (cont.)

Why?

- Changes persist
- Saves memory
- Increase speed
If we want the speed of passing by reference but don’t want our object mutated, use `const`.

```cpp
bool isOrigin(const Point& pt) {
    return pt.getX() == 0 && pt.getY() == 0;
}
```

Compiler will ensure that `pt` isn’t modified.
Speeding up the Point class:

**original:**
```cpp
double distance(Point other) const {
}
```

**faster:**
```cpp
double distance(const Point& other) const {
}
```

Another:
```cpp
Point operator+(const Point& other) const {
    return Point(_x + other._x, _y + other._y);
}
```

Note: Return type is still `value`. Why?
Recall: Point output

```cpp
ostream& operator<<(ostream& out, Point p) {
    out << "<" << p.getX() << ", " << p.getY() << ">"; // display using form <x,y>
    return out;
}
```

Here, & is required because streams cannot be copied.

Note that we don't use const since we are changing the stream by adding data.
Pointer variables

Syntax: `Point *d;`  // d is a pointer variable

d is created as a variable that stores a memory address.

So: `d = &b;`  

memory address of b

But d is not a Point! can't say d = b
Using pointer variables

2 options:

(*d). get Y();

d -> get Y();
Passing by Pointer

Point *pt = NULL

bool isOrigin(Point *pt) {
    return pt->getX() == 0 && pt->getY() == 0;
}

This is similar to passing by reference but allows you to also pass a null pointer.