CS162: Lecture 13:
Operator Overload, Friends, Pointer Review
First: HouseKeeping

“Missing a Demo: If you miss your demo with a TA, you will receive a 10 point (one letter grade) deduction to that assignment for each demo missed.” - The Syllabus.
Operator Overload

- Able to overload the assignment operator, why not others?
- Overload: more than one definition for the same function name or operator, but with a different parameter listing
- Can overload most operators, cannot overload:
  - Scope resolution operator ::
  - Pointer/Dereference (unary) *
  - Dot operator .
  - Turnary ?:
- Can’t come up with new operators, can’t change precedence, can’t change the number of operands required
- If you overload the logical operators it destroys the short circuiting aspect
Example Operator Overload

class Circle {
    private:
        float radius;
        float center_x;
        float center_y;
    public:
        float get_radius();
        float get_center_x();
        float get_center_y();
        //insert mutators here
    };

bool operator==(const Circle & c1, const Circle& c2) {
    if((c1.get_radius() == c2.get_radius()) && (c1.get_center_x() == c2.get_center_x()) && (c1.get_center_y() == c2.get_center_y()))
        return true;
    return false;
}
Friend

- Functions or classes declared with the friend keyword
- Non member function can access the protected and private members of the class if declared friend of that class
- Can have friend classes (not inheritance!) has access to the private and protected members of the friend
- Not a two way street: if declared friend in one class but not in the other the same access permissions are not provided
Friend example

class Circle {
    private:
        float radius;
        float center_x;
        float center_y;
    
    public:
        float get_radius();
        float get_center_x();
        float get_center_y();
        friend bool operator == (const Circle &, const Circle &);
        // insert mutators here

};

bool operator==(const Circle & c1, const Circle & c2) {
    if((c1.radius == c2.radius) && (c1.center_x == c2.center_x) && (c1.center_y == c2.center_y))
        return true;

    return false;
}
Pointers

A pointer is a data type that “points” to another value stored in memory.

> int x = 25;

> int* p = &x;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0003</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>x</td>
<td>25</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
</tr>
</tbody>
</table>
Using the & Operator

Line 1:
> int* x;

& - Address of

Line 2:
> int *x;

* - Contents of
Evil Laughter

```cpp
int x = 25;
int* p = &x;

cout << p << endl;
cout << x << endl;
cout << &p << endl;
cout << *p << endl;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0003</td>
<td>0000</td>
</tr>
<tr>
<td>x</td>
<td>25</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
</tr>
</tbody>
</table>
*p is an alias for x -- p is an alias for &x!

1: 0003
2: 25
3: 0000
4: 25
*p is an alias for x -- p is an alias for &x!

1 > x = x + 5;
2 > x = *p + 5;
3 > *p = *p + 5;
4 > cout << &*p << endl;

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0003</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>x</td>
<td>25</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
</tr>
</tbody>
</table>
How are Pointers really used?

- Refer to new memory reserved during program execution (dynamic memory)
- Refer to and share large data structures without making a copy of them (pass by reference)
- Specify relationships among data (linked lists, trees, graphs, etc.)