Classes and Operators

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Classes II
- Operator Overloading
Announcement
Recap
Objects

We use objects all the time (string, vector etc.)

Objects encapsulate **behaviour**

Classes allow you to define your own type as if it were built into C++
Classes

Public interface:

- Get name, suid, and number of units
- Get class year (based on units)
- Check if international/local student
Members of a struct by default public

Member of a class are by default private

```cpp
struct Student {
    std::string name;
    std::string suid;
    int unitsTaken;
};
```

```cpp
class Student {
public:
    std::string name;
    std::string suid;
    int unitsTaken;
};
```
Structs vs Classes

Members of a struct by default **public**

Member of a class are by default **private**

```cpp
class Student {
    std::string name;
    std::string suid;
    int unitsTaken;
};
```

```cpp
class Student {
    private:
        std::string name;
        std::string suid;
        int unitsTaken;
};
```
Scope Resolution

There was a lot of `std::` and `Student::` in our code.

Why?
Questions
Classes II
Designing Axess
Now we have a usable Student class.

Let’s now represent the database!

```cpp
vector<Student> database;
```
Now we have a usable Student class.

Let’s now represent the database!

```
vector<Student> database;
```
Classes

Now we have a usable Student class.

Let’s now represent the database!

```
Database db;
```
Classes

The **Database** class will internally have a vector of **Student**

Public interface:

- Add student to database
- Check if student is in database
- Get students in a single year
Classes

Example:

StudentClass
(StudentClass.pro)
C++ doesn’t know how to use operators on types defined by us:

- We can tell it how to via operator overloading.

An algorithm needed a function that could capture a local variable

- More about this next time.
Operator Overloading
Operator Overloading

Allows you to define functionality for operators on any types.

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td>!</td>
<td>,</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[ ]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>- &gt; *</td>
<td>new</td>
<td>new [ ]</td>
<td>delete</td>
<td>delete [ ]</td>
</tr>
</tbody>
</table>
Operator Overloading

Use only when overloading has an intuitive meaning:

```cpp
Set<int> numSet;
numSet += 2;
numSet += 3;
// numSet is now {2, 3}
```

Good overload of `+=` operator!
Use only when overloading has an intuitive meaning:

```cpp
Set<int> numSet;
numSet += 2;
numSet += 3;
// numSet is now {2, 3}
```
Operator Overloading

Use **only** when overloading has an intuitive meaning:

```cpp
Set<int> numSet;
numSet += 2;
umSet += 3;
numSet, 4, 5;
// numSet is now {2, 3}
```
Operator Overloading

Use **only** when overloading has an intuitive meaning:

```cpp
Set<int> numSet;
numSet += 2;
numSet += 3;
numSet, 4, 5;
// numSet is now ???
```

No intuitive understanding of what this does.
struct Point {
    int x, y;

    bool operator==(const Point& rhs) {
        return x == rhs.x && y == rhs.y;
    }
};
Operator Overloading

```cpp
struct Point {
    int x, y;
    bool operator==(const Point& rhs) {
        return x == rhs.x && y == rhs.y;
    }
};
```

Class member function. LHS is implicit this object
struct Point {
    int x, y;
    bool operator==(const Point& rhs) {
        return x == rhs.x && y == rhs.y;
    }
};

Operator Overloading
struct Point {
    int x, y;
};

bool operator==(const Point& rhs) {
    return x == rhs.x && y == rhs.y;
}
struct Point {
    int x, y;
};

bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
struct Point {
    int x, y;
};

bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
Operator Overloading

```cpp
struct Point {
    int x, y;
};

bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

Non member function. LHS is explicit first parameter.
struct Point {
    int x, y;
};

bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
Operator Overloading

Two ways to overload operators:

- Member functions
- Non-member functions
Member Functions

Just add a function named `operator@` to your class

```cpp
bool operator==(const HashSet& rhs) const;
Set operator+(const Set& rhs) const;
Set& operator+=(const ValueType& value);
```

For binary operators, accept the right hand side as an argument.

I usually name mine `rhs`.
Non-member Functions

Add a function named `operator@` outside your class.

Have it take all its operands.

```cpp
bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```
Operator Overloading

Some examples:

OperatorOverload
(OpOverload.pro)
Non-member Operators

The standard library tends to prefer this way

Allows for the lhs to be a non class type

If it needs access to internal private members, declare it in the class with the `friend` keyword!
Foreshadow...
Let’s go back for a second...

## Operator Overloading

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td>~</td>
<td>!</td>
<td>,</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
</tr>
<tr>
<td>-</td>
<td>-*</td>
<td>new</td>
<td>new</td>
<td>delete</td>
<td>delete</td>
</tr>
</tbody>
</table>
Let's go back for a second...

## Operator Overloading

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Anything curious here?
Operator Overloading

Let's go back for a second...

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>=&gt;</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;*</td>
<td>new</td>
<td>new</td>
<td>delete</td>
<td>delete []</td>
</tr>
</tbody>
</table>
Operator Overloading

Let's go back for a second...

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td>~</td>
<td>!</td>
<td>,</td>
<td>=</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
</tr>
<tr>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;&gt;=</td>
<td>[]</td>
</tr>
</tbody>
</table>

Anything curious here?
Next Time

Functions