Functions

Ali Malik
malikali@stanford.edu
Game Plan

Recap

Operator Overloading

Functions

Lambdas
Announcements
Recap
C++ doesn’t know how to use operators on types defined by us

An algorithm needed a function that could capture a local variable
## 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>
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& rhs) {
        return x == rhs.x && y == rhs.y;
    }
};
```cpp
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& 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;
}
struct Point {
    int x, y;
};

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

```c
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
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!
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>-&gt;</td>
<td>-*</td>
<td>new</td>
<td>new</td>
<td>[]</td>
<td>delete</td>
</tr>
</tbody>
</table>
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;=</td>
<td>[]</td>
</tr>
</tbody>
</table>

Anything curious here?
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>-&gt;</td>
<td>-*</td>
<td>new</td>
<td>new [ ]</td>
<td>delete</td>
<td>delete [ ]</td>
</tr>
</tbody>
</table>
Operator Overloading

Some experimentation:

FunctionOperator
(FuncitonOp.pro)
Functors
Functors

Classes which define the \( () \) operator.

Why is this useful?

- Can have state
- Customizable through constructor

Very useful for algorithms!
Remember this problem?

```cpp
std::vector<Student> StudentDatabase::studentsInYear(std::string yearToFind) {
    vector<Student> ret;

    // Can't use pred function because we need to somehow give it yearToFind...
    // std::copy_if(db.begin(), db.end(), std::back_inserter(ret), pred);

    // We'll settle for a for-loop
    for(auto student : db) {
        if(student.classLevel() == yearToFind) {
            ret.push_back(student);
        }
    }

    return ret;
```
Operator Overloading

Using functors:

StudentClass
(StudentClass.pro)
Functors let us make customizable functions!

We can pass useful information to their constructor that was not known at compile time.

But...

Kind of a Pain™
Functors

Functors let us make **customizable** functions!

We can pass **useful information** to their **constructor** that was not known at compile time.

But…

Kind of a Pain™

C++ has a solution!
Functors

Functors let us make *customizable* functions!

We can pass *useful information* to their *constructor* that was not known at compile time.

But...

Kind of a Pain™

**C++11** has a solution!
Lambdas
Lambdas

A C++11 feature that lets you make functions on the fly.

```
[capture-list](params) -> ReturnType {
  // code
};
```
Lambdas

Best learnt by example:

```cpp
auto print_int = [](int x) {
    cout << x << endl;
};

print_int(5) // outputs 5 to console
```
Lambdas

Best learnt by example:

```cpp
vector<int> v{3, 1, 4, 1, 5};
std::sort(v.begin(), v.end(),
          [](int i, int j) -> bool { return i > j; });

// sorts vector in decreasing order
```
Most modern languages have lambdas in some form!

`lessThanPy = lambda x, y: x < y`

`const lessThanJs = (x, y) => {return x < y;};`

`Comparator lessThanJava = (x, y) -> return x < y;`

`auto lessThanCpp = [](int i, int j) -> bool { return i < j;};`
Questions
Lambda Captures

A C++11 feature that lets you make functions on the fly.

```cpp
[capture-list](params) -> ReturnType {
    // code
};
```
Lambda Captures

A C++11 feature that lets you make functions on the fly.

```
[capture-list](params) -> ReturnType {
  // code
};
```

What is this for?
std::vector<Student> StudentDatabase::studentsInYear(std::string yearToFind) {
    vector<Student> ret;

    // Can't use pred function because we need to somehow give it yearToFind...
    // std::copy_if(db.begin(), db.end(), std::back_inserter(ret), pred);

    // We'll settle for a for-loop
    for(auto student : db) {
        if(student.className() == yearToFind) {
            ret.push_back(student);
        }
    }

    return ret;
}
Lambda Captures

You can capture available variables to use in the lambda

\[ byValue, \ & byReference \]

You can also capture all currently available variables:

\[=\] // By value

\[\&\] // By reference

This will only capture the ones used inside the function.
How Does This Work?
How Lambdas Work?

```cpp
class SomeName {
public:
    SomeName(capture-list) {
        // set each private member to
        // thing in capture list
    }

    ReturnType operator() (params) {
        // code
    }

private:
    // create private member for each
    // thing in capture-list
};
```
Complex Data Structures
std::queue

A First-In/First-Out (FiFO) data structure

```cpp
template<
    class T,
    class Container = std::deque<T>
>
class queue;
```
std::queue

Needs two template types to be constructed:

**Template parameters**

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

- **Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer. Additionally, it must provide the following functions with the usual semantics:
  - back()
  - front()
  - push_back()
  - pop_front()

The standard containers `std::deque` and `std::list` satisfy these requirements.
A priority queue lets us get the earliest event in the sequence!

```cpp
std::priority_queue

// Defined in header <queue>

template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
>
    class priority_queue;
```
std::priority_queue

Needs three template types to be constructed:

**Template parameters**

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as `std::vector::value_type`. *(since C++17)*
- **Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`, and its iterators must satisfy the requirements of `RandomAccessIterator`. Additionally, it must provide the following functions with the usual semantics:
  - `front()`
  - `push_back()`
  - `pop_back()`

The standard containers `std::vector` and `std::deque` satisfy these requirements.

- **Compare** - A `Compare` type providing a strict weak ordering
How do we make a priority_queue of names ordered by last name?

```cpp
// Function to compare two names by last name
auto cmpFn = ...;

// Create priority queue
std::priority_queue<
    vector<string>,
    vector<vector<string>>,
    decltype(cmpFn)> queue(cmpFn);
```
Next Time

ParticleSimulator