Operator Overloading

Cristian Cibils
(ccibils@stanford.edu)

Announcements

- Assignment 3 goes out Today!
- Due Tuesday, May 26th
- Even if you did the first 2 assignments, I recommend doing the third since it is really cool
- You get to build a multidimensional binary search tree

Why Operator Overloading?

Let's say we were using our good old point class again:

```
Point a(1, 2);
Point a(2, 1);
```

Why Operator Overloading?

I want to be able to add points together and produce a result:

```
Point a(1, 1);
Point b(1, 2);
Point c = a + b;
```

Why Operator Overloading?

Unfortunately, the compiler doesn't know how to add points:

```
main.cpp:6: error: no match for
'operator+' (operand types are 'Point'
and 'Point')
     Point c = a + b;
                   ^
```

Operator Overloading

Operator Overloading allows us to define the meaning of "+" and other operators when used on a type we defined

What Can I Overload?

Here's a sample of some of the operators you can overload (there are many more):

Arithmetic	`+, -, *, /, %`
Comparison	`!=, ==, <, <=, >, <=`
Access	`[], *, ->`
Stream	`<<, >>`
Scary	`new, delete, ','`

Operator Overloading

Let’s start by overloading a simple operator: the == operator.

Operator Overloading

- Two `Point`s are equal if there `x` and `y` coordinates are the same.
- How can we write this as an operator overload?
- Two ways:
  - Member function syntax
  - Free function syntax

Operator Overloading

- Two `Points` are equal if there `x` and `y` coordinates are the same.
- How can we write this as an operator overload?
- Two ways:
  - Member function syntax
  - Free function syntax

Member Function Syntax

- The preferred way to overload an operator is to add a member function with a special name (`operator==` in this case)
- The left hand side of the operator is the object whose member function is called
- The member function takes one argument, which is the right hand side of the operator
- By convention, `operator==` returns a `bool`

Member Function Syntax

```
Class Point { // abbreviated
   double x, y;
    bool operator==(const Point& rhs) {
       return (x == rhs.x  && y == rhs.y);
    }
};
```

Member Function Syntax

```
Class Point { // abbreviated
   bool operator==(const Point& rhs) {
      return (x == rhs.x  && y == rhs.y);
   }
};
Point p1(3, 2);
Point p2(3, 2);
if (p1 == p2)
   cout << “Points are equal!” << endl;
```

Member Function Syntax

```
Class Point { // abbreviated
   bool operator==(const Point& rhs) {
      return (x == rhs.x  && y == rhs.y);
   }
};
Point p1(3, 2);
Point p2(3, 2);
if (p1.operator==(p2))
   cout << “Points are equal!” << endl;
```

Operator Overloading

- Two `Points` are equal if there `x` and `y` coordinates are the same.
- How can we write this as an operator overload?
- Two ways:
  - Member function syntax
  - Free function syntax

Free Function Syntax

```
bool operator==(Point l, Point r) {
   return l.x == r.x && l.y == r.y;
}
Point p1(1, 2);
Point p2(1, 2);
if (p1 == p2)
   cout << “Points are equal!” << endl;
```

Free Function Syntax

```
bool operator==(Point l, Point r) {
   return l.x == r.x && l.y == r.y;
}
Point p1(1, 2);
Point p2(1, 2);
if (operator==(p1, p2))
   cout << “Points are equal!” << endl;
```

Free Function Syntax

Here’s a better example of when you need free function syntax -- multipliplying a point by a scalar.

Free Function Syntax

```
Point operator*(double l, Point r) {
   Point result(l * r.x, l * r.y);
   return result;
}
Point p(1, 1);
Point result = 5 * p;
result.print(); // prints (5, 5)
```

Operator Overloading

Lets add some operators to our point

Friendly Note

- As you all know, no function outside of a class can access that class's private functions/variables
  - This is a good thing! (most of the time)
- Sometimes, you want a function or class to access the private parts of another class. In these cases, use the **friend** keyword

Friendly Note

```
Class Foo {
private:
   int x;
};

//NOT ALLOWED! Cannot access private
//member variable
void doStuff(Foo& f) {
   f.x = 5;
}
```

Friendly Note

```
Class Foo {
private:
   friend doStuff(Foo& f);
   int x;
};

//ALLOWED! doStuff is now a friend function
void doStuff(Foo& f) {
   f.x = 5;
}
```

Friendly Note

```
Class Foo {
private:
   friend class Boo; //makes boo a friend class
   int x;
};

// Boo can now access private parts of Foo
Class Boo {
};
```

A Word of Caution

Operator Overloading should only be used when the meaning of the operator is obvious:

```
Point p1, p2;
Point result = p1 + p2;
Point scaledResult = 5 * result;

string one = “hi”, two = “hello”;
bool stringsSame = (one == two);
```

A Word of Caution

Operator overloading can be abused, and the results are scary:

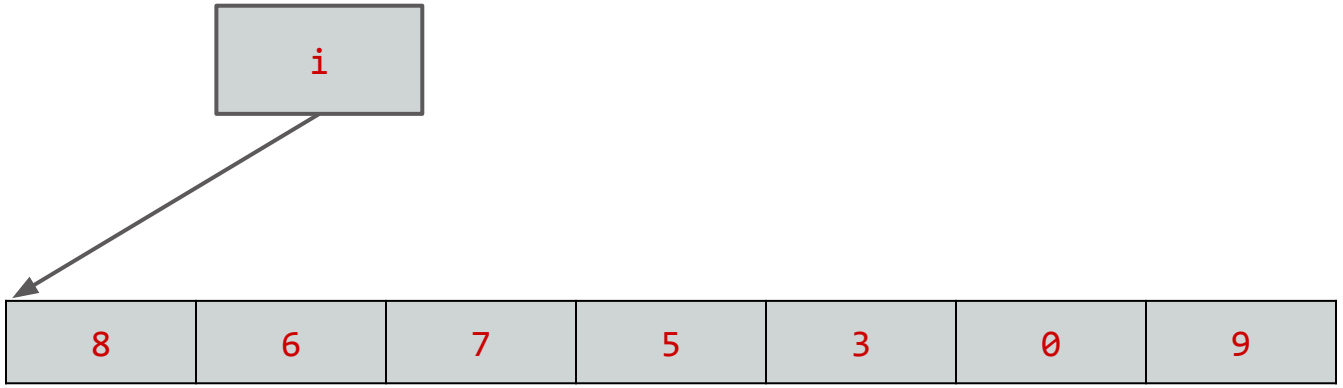
```
// This works with a Stanford vector
Vector<int> v;
1, 2, v, 4;
cout << v[0] << endl; // ?
```

A Slightly More Advanced Overload

Let’s try adding some overloads to our vector type.

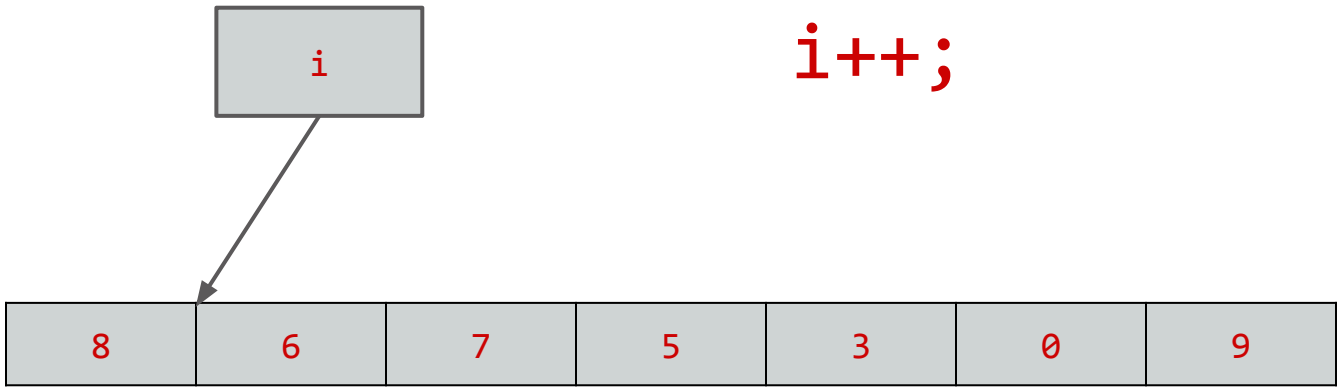
Vector Iterator

Remember how we could just use a pointer for a `vector`'s iterator?



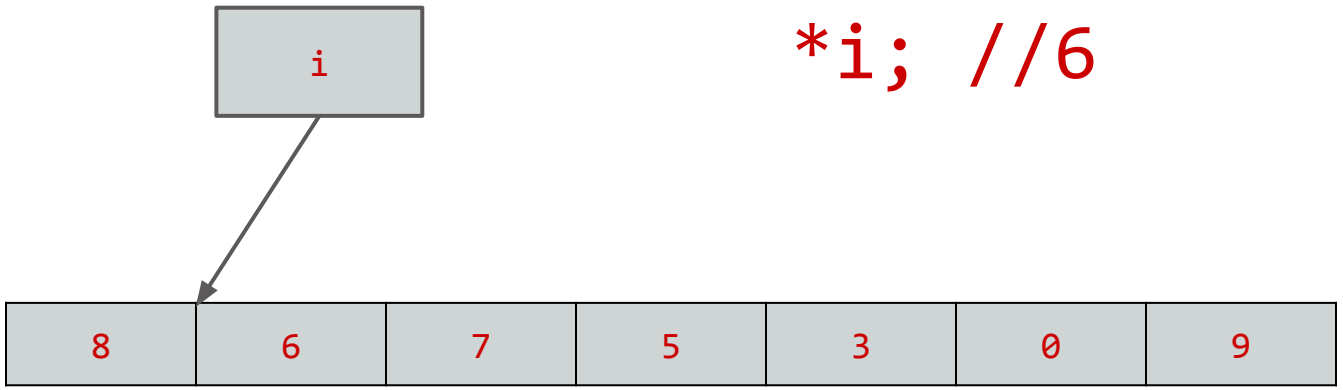
Vector Iterator

Remember how we could just use a pointer for a `vector`'s iterator?



Vector Iterator

Remember how we could just use a pointer for a `vector`'s iterator?

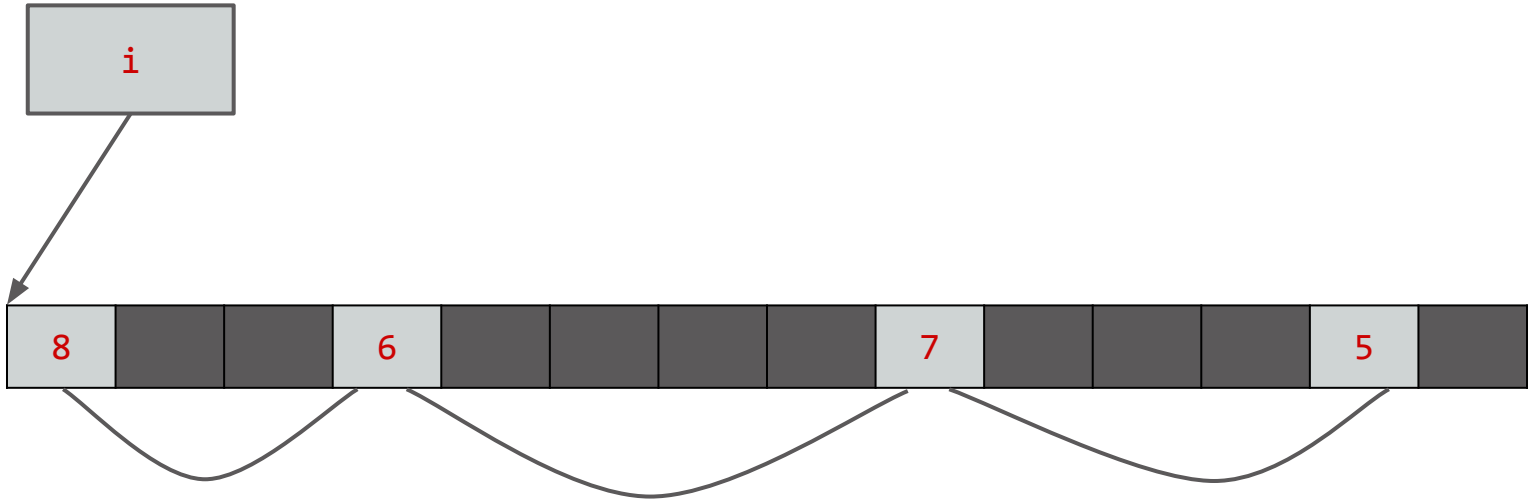



```
*i; //6
```

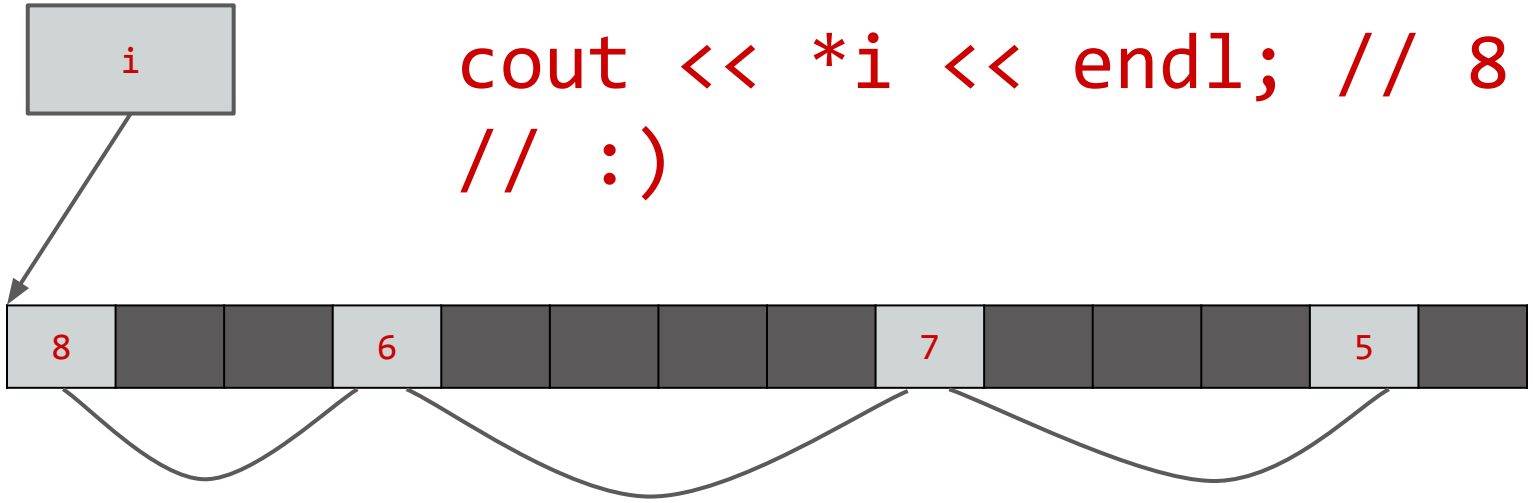
Linked List Iterator

Can we do the same thing for linked lists?

Linked List Iterator

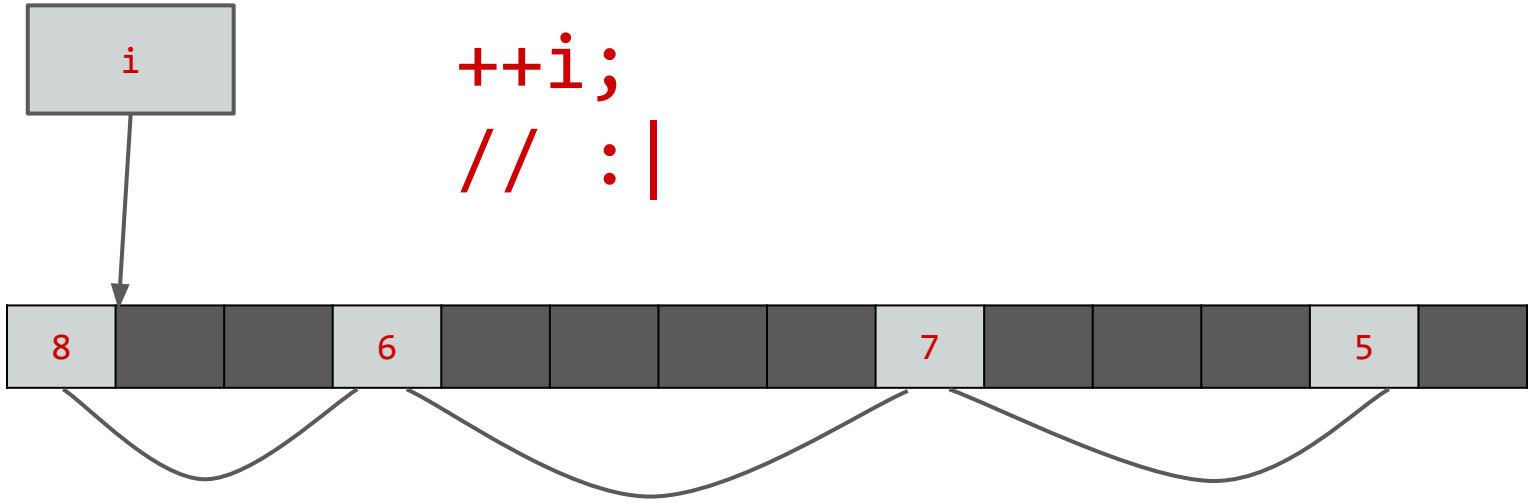


Linked List Iterator

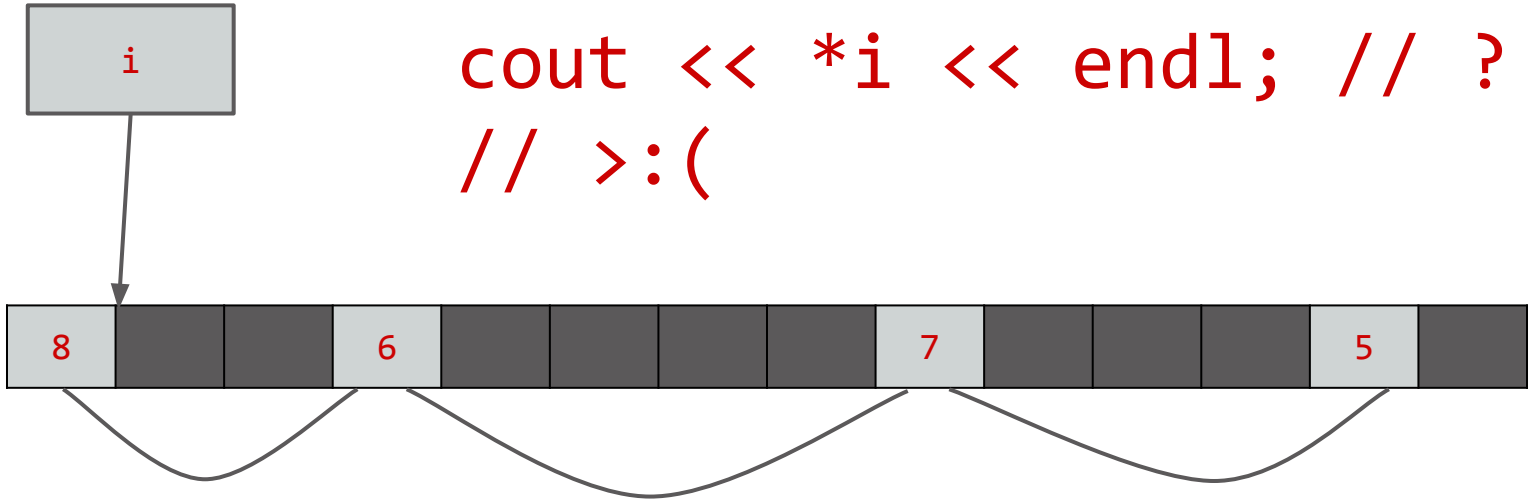



```
cout << *i << endl; // 8
// :)
```

Linked List Iterator



Linked List Iterator




```
cout << *i << endl; // ?
// >:(
```

++i vs i++

```
int i = 1;    // i = 1
int j = i++; // j = 1, i = 2
int k = ++i; // k = 3, i = 3
```

++i vs i++

```
// create a vector of ints called nums
vector<int>::iterator i = nums.begin();

vector<int>::iterator j = i++;
vector<int>::iterator k = ++i;
```

++i vs i++

- To differentiate **preincrementation** with **postincrementation**, preincrementation takes in no parameters, and postincrementation takes in one parameter, a dummy int

++i vs i++

```
class foo {
  // Define ++foo
  foo operator++() {

  }
  // Define foo++
  foo operator++(int) {

  }
}
```

Linked List Iterator

Let’s make our Linked List iterator

Last Overload

- One last operation I want to overload is the **->** operator
- This one acts a little strange since C++ calls the -> operator on whatever is returned from the -> operator

```
itr->x == (itr.operator->())->x;
```