Operator Overloading

See also: Chapter 9 (209-217), Chapter 16

Simplified SpreadsheetCell (for doubles)

class SCell
{
  public:

    SCell();

    double getValue() const;
    void setValue(double inValue);

  protected:

    double mValue;
};

Goal: Facilitate easy addition of cells
Attempt #1: add() method

Solution: Create a method that lets you perform addition on one SCell, given another as input.

```cpp
const SCell SCell::add(const SCell& cell) const
{
    SCell newCell;
    newCell.set(mValue + cell.mValue);

    return newCell; // return a copy
}
```

```cpp
SCell cellOne(6);
SCell cellTwo(3);
SCell cellThree = cellOne.add(cellTwo);
```
Evaluating the First Approach

<table>
<thead>
<tr>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Simple implementation</td>
<td>• Clumsy usage syntax</td>
</tr>
<tr>
<td>• Straight-forward usage</td>
<td>• Can't add constant vals (0.4)</td>
</tr>
<tr>
<td>• Implies no behaviors</td>
<td>• Requires documentation</td>
</tr>
<tr>
<td></td>
<td>• Non-standard, non-generic</td>
</tr>
</tbody>
</table>

We can support adding constant values by adding additional constructors and taking advantage of *implicit conversions*. 
Attempt #2: add() with implicit conversions

class SCell
{
  public:
    SCell();
    SCell(double inValue) { mValue = inValue; }
    explicit SCell(const string& inValue); // don't allow
    [etc.]
};

Now we can do this:

SCell cellFour = cellThree.add(0.5); // Sweet!

But we still can't do this:

SCell cellFive = (0.5).add(cellFour); // BUG!
Attempt #3: Overloaded operator+ as a Method

We can allow familiar syntax by *overloading* operator+ to be able to add SCells:

```cpp
class SCell
{
    public:
        [etc.]
        
        const SCell operator+(const SCell& cell) const;
    };

const SCell SCell::operator+(const SCell& cell) const
{
    SCell newCell(mValue + cell.mValue);
    return newCell;
}
```
Now we can do this:

```java
SCell cellThree = cellOne + cellTwo;
```

And this:

```java
SCell cellFour = cellThree + 0.5;
```

But we still can't do this:

```java
SCell cellFive = 0.2 + cellFour;  // BUG!
```

When overloading an operator as a method of a class, the object in question is the left-hand side of the operation.
Attempt #4: Overloaded operator+ as a Global Function

// entirely outside of class SCell:
const SCell operator+(const SCell& lhs, const SCell& rhs) {
    SCell newCell;
    newCell.setValue(lhs.getValue() + rhs.getValue());

    return newCell;
}

It's often helpful to use the friend keyword when implementing overloaded operators as global functions:

class SCell {
    public:
        friend const SCell operator+(const SCell& lhs, const SCell& rhs);
    [etc.]
}
What You *Can Do*

Key concept: Be careful to implement the behavior that one would *expect* for the given operator, even though C++ doesn't require you to.

- Behave however you want (operator* can do division)
- Take whatever types you want (myCell + myGrid???)
- Return whatever you want (myCrazyDelicious = myRedVine + myMrPibb)
- Overload pretty much anything (including op[], op*, op())
What You Can't Do

• Add new operators

• Overload . (dot), :: (scope resolution), sizeof, ?:, etc.

• Change the number of operands (myCell++2)

• Change order of evaluation (i * j + k)

• Change operators for built-in types (except new and delete)

• Infer one operator from another (you don't get += for free)
Adding in operator+=

class SCell
{
    public:
        friend const SCell operator+(const SCell...;

        // for this one, we want it to be a method
        SCell& operator+=(const SCell& rhs);

    [etc.]
};

SCell& SCell::operator+=(const SCell& rhs)
{
    setValue(mValue + rhs.mValue);
    return (*this);
}
Overloading Comparison Operators

Implement comparison operators as friend functions that return a bool:

class SCell {
    public:
        friend bool operator<(const SCell& lhs, const SCell& rhs);
    [etc]
};

bool operator<(const SCell& lhs, const SCell& rhs) {
    return (lhs.getValue() < rhs.getValue());
}

// implement >= in terms of op<
bool operator>=(const SCell& lhs, const SCell& rhs) {
    return !(lhs < rhs);
}
<table>
<thead>
<tr>
<th>Operator</th>
<th>Name or Category</th>
<th>Method or Global Friend Function</th>
<th>When to Overload</th>
<th>Sample Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator+</td>
<td>Binary arithmetic</td>
<td>Global friend function recommended</td>
<td>Whenever you want to provide these operations for your class</td>
<td>friend const T operator+(const T&amp;, const T&amp;);</td>
</tr>
<tr>
<td>operator-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator-</td>
<td>Unary arithmetic and bitwise operators</td>
<td>Method recommended</td>
<td>Whenever you want to provide these operations for your class</td>
<td>const T operator-() const;</td>
</tr>
<tr>
<td>operator+</td>
<td>Increment and decrement</td>
<td>Method recommended</td>
<td>Whenever you overload binary + and -</td>
<td>T&amp; operator++; const T operator++(int);</td>
</tr>
<tr>
<td>operator--</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator=</td>
<td>Assignment operator</td>
<td>Method required</td>
<td>Whenever you have dynamically allocated memory in the object or want to prevent assignment, as described in Chapter 9</td>
<td>T&amp; operator=(const T&amp;);</td>
</tr>
<tr>
<td>operator+=</td>
<td>Shorthand arithmetic operator assignments</td>
<td>Method recommended</td>
<td>Whenever you overload the binary arithmetic operators</td>
<td>T&amp; operator+=(const T&amp;);</td>
</tr>
<tr>
<td>operator-=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator*/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator/=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>Binary bitwise</td>
<td>Global friend</td>
<td>Whenever you want</td>
<td>friend const T operator&lt;&lt;(const T&amp;, const T&amp;);</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Function Recommended</td>
<td>When to Provide</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------------</td>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>operator&gt;&gt;</code></td>
<td>Shorthand bitwise operator assignments</td>
<td>Method recommended</td>
<td>Whenever you overload the binary bitwise operators</td>
<td><code>T&amp; operator&lt;&lt;=(const T&amp;);</code></td>
</tr>
<tr>
<td><code>operator&lt;&lt;</code></td>
<td>I/O stream operators (insertion and extraction)</td>
<td>Global friend function recommended</td>
<td>Whenever you want to provide these operations</td>
<td><code>friend ostream &amp;operator&lt;&lt;(ostream&amp;, const T&amp;);</code>&lt;br&gt;<code>friend istream &amp;operator&gt;&gt;(istream&amp;, T&amp;);</code></td>
</tr>
<tr>
<td><code>operator!</code></td>
<td>Boolean negation operator</td>
<td>Member function recommended</td>
<td>Rarely; use <code>bool</code> or <code>void*</code> conversion instead</td>
<td><code>bool operator!() const;</code></td>
</tr>
<tr>
<td><code>operator&amp;&amp;</code></td>
<td>Binary Boolean operators</td>
<td>Global friend function recommended</td>
<td>Rarely</td>
<td><code>friend bool operator&amp;&amp;(const T&amp; lhs, const T&amp; rhs);</code>&lt;br&gt;`friend bool operator</td>
</tr>
<tr>
<td><code>operator[]</code></td>
<td>Subscripting (array index)</td>
<td>Method required</td>
<td>When you want to support subscripting</td>
<td><code>E&amp; operator[](int);</code></td>
</tr>
<tr>
<td><strong>operator</strong></td>
<td>Description</td>
<td>Required/Mandatory</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>--------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td><strong>operator()</strong></td>
<td>Function call operator</td>
<td>Method required</td>
<td>When you want objects to behave like function pointers</td>
<td>Return type and arguments can vary; see examples in this chapter</td>
</tr>
<tr>
<td><strong>operator new</strong></td>
<td>Memory allocation routines</td>
<td>Method recommended</td>
<td>When you want to control memory allocation for your classes (rarely)</td>
<td>void* operator new(size_t size) throw(bad_alloc); void* operator new[](size_t size) throw(bad_alloc);</td>
</tr>
<tr>
<td><strong>operator delete</strong></td>
<td>Memory deallocation routines</td>
<td>Method recommended</td>
<td>Whenever you overload the memory allocation routines</td>
<td>void operator delete(void* ptr) throw(); void operator delete[](void* ptr) throw();</td>
</tr>
<tr>
<td><em><em>operator</em> / operator-&gt;</em>*</td>
<td>Dereferencing operators</td>
<td>Method required for operator-&gt; Method recommended for operator*</td>
<td>Useful for smart pointers</td>
<td>E&amp; operator*() const; E* operator-&gt;() const;</td>
</tr>
<tr>
<td><strong>operator&amp;</strong></td>
<td>Address-of operator</td>
<td>N/A</td>
<td>Never</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>operator-&gt;</strong>*</td>
<td>Dereference pointer-to-member</td>
<td>N/A</td>
<td>Never</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>operator,</strong></td>
<td>Comma operator</td>
<td>N/A</td>
<td>Never</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>operator type()</strong></td>
<td>Conversion, or cast, operators (separate per type)</td>
<td>Method required</td>
<td>When you want to provide conversions from your class to other types</td>
<td>operator type() const;</td>
</tr>
</tbody>
</table>
Prefix ++ versus Postfix ++

class SCell
{
    public:
        SCell& operator++(); // prefix
        const SCell operator++(int); // postfix
};

Why does the postfix version return a const?

Why does it take an int?
Overloading the stream operators (<< and >>)

class SCell {
    public:
        friend ostream& operator<<(ostream& out, const SCell& cell);
        friend istream& operator>>(istream& in, SCell& cell);
    
    ostream& operator<<(ostream& out, const SCell& cell) {
        out << "[ " << cell.getValue() << " ]";
        return out;
    }

    // WARNING: this won't work on decimals
    istream& operator>>(istream& in, SCell& cell) {
        string temp;
        in >> temp; // ":[
        in >> temp;
        cell.setValue(temp);
        in >> temp; // "]"

        return in;
    }
}
Overloading operator[]

// for myArray[3] = 3;
int& operator[](int x);

// for cout << myArray[2]; (where myArray is const)
const int& operator[](int x) const;

The version called depends on the type of the object, so you better make them do the same thing!

Open a new world of indexing possibilities by using non-int op[]:

string& operator[](string inKey);
const string& operator[](string inKey) const;
Overloading to Provide Automatic Conversion

SCell myCell(7.2);
string str = myCell; // BUG!
string str = (string)myCell; // BUG!

class SCell {
   public:
      operator string() const; // note lack of return type
};

SCell::operator string() const
{
   string retVal;
   retVal += getValue();
   return retVal;
}

Warning: This feature can work against implicit conversion!
Operator Overloading and the STL

The priority_queue class automatically orders objects based on their priority, but you need to specify how that priority is determined.

class Show
{
    public:
        Show(int inPriority) { mPriority = inPriority; }
        int getPriority() { return mPriority; }
        friend bool operator<(const Show& lhs, const Show& rhs);
    protected:
        int mPriority;
};

bool operator<(const Show& lhs, const Show& rhs) {
    return lhs.mPriority < rhs.mPriority;
}
int main(int argc, char** argv)
{
    Show s1(1);
    Show s2(2);
    Show s3(3);

    priority_queue<Show> myQueue;
    myQueue.push(s1);
    myQueue.push(s2);
    myQueue.push(s3);

    cout << "The top Show is " <<
         myQueue.top().getPriority() << endl;
}

Functors – Gateway to STL Algorithms

You can overload operator(), the function call operator, to create a functor. Functors allow you to use an object as a function pointer.

class Div
{
    public:
        Div(bool inDivZero) { mDivZero = inDivZero; }

        int operator() (int inNum, int inDenom);

    protected:
        bool mDivZero;
};
int Div::operator() (int inNum, int inDenom) 
{ 
    if (inDenom == 0 && mDivZero) {
        return 0;
    } else if (inDenom == 0) {
        throw std::exception();
    } else {
        return (inNum / inDenom);
    }
}

Now we can do this:

int main()
{
    Div myDivider(true);

    cout << "3/2 is " << myDivider(3, 2) << endl;
}
Functors: So What?

At first, it seems like the functionality of a functor could be implemented just as easily with a method (divide()), but the magic comes when you have a templatized method that uses a functor as a callback.

Example: Priority Queue based on Greater Than:

```
priority_queue<int, vector<int>, greater<int> > myQueue;
```

greater<int> is a predefined STL functor, which lives in include <functional>