

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.

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

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

```
SCell cellOne(6);
```

```
SCell cellTwo(3);
```

```
SCell cellThree = cellOne.add(cellTwo);
```

Evaluating the First Approach

Strengths	Weaknesses
<ul style="list-style-type: none">• Simple implementation• Straight-forward usage• Implies no behaviors	<ul style="list-style-type: none">• Clumsy usage syntax• Can't add constant vals (0.4)• Requires documentation• Non-standard, non-generic

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:

```
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:

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

And this:

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

But we still can't do this:

```
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));
}
```

Operator	Name or Category	Method or Global Friend Function	When to Overload	Sample Prototype
operator+ operator- operator* operator/ operator%	Binary arithmetic	Global friend function recommended	Whenever you want to provide these operations for your class	friend const T operator+(const T&, const T&);
operator- operator+ operator~	Unary arithmetic and bitwise operators	Method recommended	Whenever you want to provide these operations for your class	const T operator-() const;
operator++ operator--	Increment and decrement	Method recommended	Whenever you overload binary + and -	T& operator++(); const T operator++(int);
operator=	Assignment operator	Method required	Whenever you have dynamically allocated memory in the object or want to prevent assignment, as described in Chapter 9	T& operator=(const T&);
operator+= operator-= operator*/ operator/= operator%=	Shorthand arithmetic operator assignments	Method recommended	Whenever you overload the binary arithmetic operators	T& operator+=(const T&);
operator<<	Binary bitwise	Global friend	Whenever you want	friend const T operator<<(const T&, const

operator>> operator& operator operator^	operators	function recommended	to provide these operations	T&);
operator<< = operator>> = operator&= operator = operator^=	Shorthand bitwise operator assignments	Method recommended	Whenever you overload the binary bitwise operators	T& operator<<=(const T&);
operator< operator> operator<= operator>= operator==	Binary comparison operators	Global friend function recommended	Whenever you want to provide these operations	friend bool operator<(const T&, const T&);
operator<< operator>>	I/O stream operators (insertion and extraction)	Global friend function recommended	Whenever you want to provide these operations	friend ostream &operator<<(ostream&, const T&); friend istream &operator>>(istream&, T&);
operator!	Boolean negation operator	Member function recommended	Rarely; use bool or void* conversion instead	bool operator!() const;
operator&& operator	Binary Boolean operators	Global friend function recommended	Rarely	friend bool operator&&(const T& lhs, const T& rhs); friend bool operator (const T& lhs, const T& rhs);
operator[]	Subscripting (array index)	Method required	When you want to support subscripting:	E& operator[](int);

	operator		in array-like classes	<code>const E& operator[](int) const;</code>
<code>operator()</code>	Function call operator	Method required	When you want objects to behave like function pointers	Return type and arguments can vary; see examples in this chapter
<code>operator new</code> <code>operator new[]</code>	Memory allocation routines	Method recommended	When you want to control memory allocation for your classes (rarely)	<code>void* operator new(size_t size) throw(bad_alloc);</code> <code>void* operator new[](size_t size) throw(bad_alloc);</code>
<code>operator delete</code> <code>operator delete[]</code>	Memory deallocation routines	Method recommended	Whenever you overload the memory allocation routines	<code>void operator delete(void* ptr) throw();</code> <code>void operator delete[](void* ptr) throw();</code>
<code>operator*</code> <code>operator-></code>	Dereferencing operators	Method required for <code>operator-></code> Method recommended for <code>operator*</code>	Useful for smart pointers	<code>E& operator*() const;</code> <code>E* operator->() const;</code>
<code>operator&</code>	Address-of operator	N/A	Never	N/A
<code>operator->*</code>	Dereference pointer-to-member	N/A	Never	N/A
<code>operator,</code>	Comma operator	N/A	Never	N/A
<code>operator type()</code>	Conversion, or cast, operators (separate per type)	Method required	When you want to provide conversions from your class to other types	<code>operator type() const;</code>

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 templated 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>