Operating Overloading

Most languages allow programmers to define custom data types such as **`enum`**s, **`struct`**s, and **`class`**es. C++ in particular—for better for worse—allows them to overload the built-in operators for these custom types. The intent is to encourage programmers to integrate their data types into the C++ language as fully as possible.

Last week we overloaded the assignment operator for the **`string`** class. If we didn't, then the compiler would synthesize one for us, and it would do the wrong thing. We're familiar enough with the **real** C++ **`string`** class to know that we can concatenate them using **`+`**, extend them using **`+=`**, access individual characters via **`[]`**, print them using **`<<`**, and compare them using **`==`**, **`!=`**, **`<`**, **`<=`**, **`>`**, and **`>=`**. All of these are available because the C++ **`string`** class overloads all of these operators for us.

We'll continue working with our own version of **`string`** as if the built-in one doesn't exist. Just to illustrate syntax, we'll take last week's string and add support for **`[]`**, **`+`** and **`+=`**. Here's our first shot at the updated interface:

```
class string {

   public:
      string(const char *str = "");
      string(const string& original);
      const string& operator=(const string& rhs);
      ~string();

      int length() const { return front - end; }
      char& operator[](int i) { return front[i]; }
      const char& operator[](int i) const { return front[i]; }
      const string operator+(const string& other) const;
      const string& operator+=(const string& rhs);

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};
```

Both implementations of **`operator[]`** are farcically obvious. Why we need two versions is probably less obvious.

Consider the following snippet of code:

```
string bestFriend = "Mark";
bestFriend[3] = 'y';
```

This second line works, because it's really a call to the non-**const** version of **string::operator[]**, which provides a reference to the fourth of four visible characters behind the scenes. (Notice that the return type of the non-**const** version is **char&**, not **char**.) It's because **bestFriend** is a non-**const string** that the non-**const** version gets called.

Here's another example where a **string** is reversed in place:

```
static void reverse(string& str)
{
   int lh = 0;
   int rh = str.length() – 1; // just pretend there's a length method ☺

   while (lh < rh) {
      char ch = str[rh];   // compiles to: char ch = str.operator[](rh);
      str[rh] = str[lh];   //              str.operator[](rh) = str.operator[](lh);
      str[lh] = ch;        //              str.operator[](lh) = ch;
   }
}
```

Because **str** is a reference to a mutable **string**, the compiler chooses the non-**const** version of **operator[]**. That's what we want—the one that returns a reference some character held behind closed doors so it can be assigned to something else.

The **const** version is called whenever the receiving **string** object is **const**, as with:

```
static int countOccurrences(const string& str, char letter)
{
   int count = 0;
   for (int i = 0; i < str.length(); i++) {
      if (str[i] == ch) // compiles to str.operator[](i) == ch
         count++;
   }
}
```

Note the return type of the **const** version is a character value, not a character reference. It's the return type that prevents the use of **operator[]** to change characters behind the scenes.

operator+ and **operator+=** are more involved, of course. Here's the implementation of each:

```
const string string::operator+(const string& other) const
{
   char buffer[this->length() + other.length() + 1];
   strcpy(buffer, this->front);
   strcpy(buffer + this->length(), other.front);
   return buffer;    // char * constructor is called to create return value
}
```

```
const string& string::operator+=(const string& rhs)
{
   *this = *this + rhs; // not the most efficient, but gets the job done
   return *this;
}
```

The **operator+** method builds the concatenation as an in-place C string, and then constructs the C++ **string** version of it as it returns. We have to return a **string** instead of a **string&**, because the return value can't be stored in any memory that existed before the method was called. (It is true that we prefer to return a reference if at all possible, because it saves the construction and destruction of a temporary object.)

Our implementation of **operator+=** frames the self-concatenation in terms of the previously written **operator=** and **operator+**. There are more efficient ways to implement this, but I'm more interested in understanding the signature of the method than I am in the implementation of it. Note that this one **does** return a reference instead of a copy.

And now some code to illustrate:

```
string sodium("Na");
string chlorine("Cl");
string tablesalt = sodium + chloride;

string singular("thing");
string plural = singular + "s";
```

Note that **tablesalt** is copy constructed out of whatever **sodium.operator+(chloride)** returns. Quite predictably, **tableSalt** surrounds **"NaCl"** after the copy constructor is through. No surprises here.

The right hand side of the fifth line translates to **singular.operator+("s")**. Those particularly observant will notice that **operator+** accepts a **const string&**, not a **const char ***. If there had been a **const char *** version of **operator+**, then it would certainly have called it, but since there isn't, C++ will create a temporary **string** object—one that can be caught by **const** reference—so that the one **operator+** method we **do** have can be used. C++ does this for all methods—it will create temporaries out of slightly type-mismatched data, provided there's some **public**, non-**explicit** constructor that can do it. Because there's a **string::string(const char *)** constructor, C++ will invoke that constructor here so that **operator+** works.

Some relevant asides:

- The compiler will only create temporaries if there's a single constructor that can be used. It won't cascade through a sequence of two or more constructors to create temporaries around temporaries so that the wrong type can eventually be converted to the right one.

- The compiler will only create nameless temporaries if they're caught by **const** reference (as opposed to non-**const** reference.) C++ isn't interested in creating temporaries to be caught by mutable reference, because it doesn't want the client to think that the original data is being updated when in fact an anonymous object is instead.
- C++ is happy to create temporaries to be caught as arguments, but it will not create temporaries to receive messages.

This last point is worth talking about some more. What it's saying is that the second line of the following will compile, but the third will not:

```
string tea = "tea";
string teapot = tea + "pot";     // effectively teapot(tea.operator+("pot"));
string icedtea = "iced " + tea;  // tries to be icedtea("iced ".operator(tea));
```

Some languages—C# and Python come to mind—would cope with that last line by constructing a **string** object around "iced " that would exist long enough to hear and respond to the **operator+** call. C++ is not one of them.

Recognizing that some programmers are bugged by the asymmetry of it (**tea + "pot"** **good**, **"iced " + tea** bad), many will opt to implement **operator+** not as a one argument method, but as a two-argument global function, as with:

```
class string {

   public:
      string(const char *str = "");
      string(const string& original);
      const string& operator=(const string& rhs);
      ~string();

      int length() const { return front – end; }
      char& operator[](int i) { return front[i]; }
      const char& operator[](int i) const { return front[i]; }
      const string& operator+=(const string& rhs);

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};

// global function
const string operator+(const string& first, const string& second);
```

The problem with this approach is that the implementation of **operator+**—no longer a privileged **string** method—would be blocked from touching **first**'s and **second**'s **private** data. So, by trying to accommodate the client's desire for symmetry, the implementation of **operator+** is being penalized. Not. What. We. Want.

Fortunately, C++ allows classes to grant **friend**ship to functions (and outside methods, and even entire classes). By granting **friend**ship, you're allowing the implementation of global functions and outside methods to access and even update the **private** data it'd normally be blocked from seeing. Here's the updated **.h** file:

```
class string {

   friend const string operator+(const string& first, const string& second);

   public:
      string(const char *str = "");
      string(const string& original);
      const string& operator=(const string& rhs);
      ~string();

      char& operator[](int i) { return front[i]; }
      char operator[](int i) const { return front[i]; }
      const string& operator+=(const string& rhs);

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};
```

You plant the prototype of the global function inside the class itself, and decorate it with the **friend** keyword. This declaration alone serves as the global prototype, so there's no need to repeat it anywhere else in the **.h** file. The placement of the **friend** function prototype is immune to the **public** and **private** access modifiers, regardless of where it is.

Now the implementation of **operator+** would look like this:

```
const string operator+(const string& first, const string& second)
{
   char buffer[first.length() + second.length() + 1];
   strcpy(buffer, first.front);
   strcpy(buffer + first.length(), second.front);
   return buffer;
}
```

You implement it as a traditional function without repeating the **friend** keyword. Had the **operator+** function not been marked as a **friend**, its implementation would have been blocked from accessing the **front** fields and forced to be a normal client of the **string** class.

Here are a few more nuggets about friendship:

- Friendship can be granted to other methods, as with:

```
class Vector {

   friend const Matrix& Matrix::operator*=(const Vector& coeffs);
   …
};
```

  This means the implementation of **`Matrix::operator*=`** can access the **`private`** data of the **`Vector`** that's passed in by reference. When two classes are as tightly coupled as **`Matrix`** and **`Vector`** are, it's not uncommon for each to grant friendship to parts of the other.

- Friendship can even be granted to entire classes, as with:

```
class Vector {

   friend class Matrix;
   …
};

class Matrix {

   friend class Vector;
};
```

  If **`Matrix`** and **`Vector`** are really being implemented in terms of each other, then there's a case for making them mutual **`friend`**s.

- In general, you should be wary of granting **`friend`**ship unless there's a spectacular reason to. It's only when a block of code outside the class is really being implemented for the class itself that you should consider using **`friend`**. In the case of **`operator+`**, we took **`operator+`** out of the **`string`** class to make the **`string`** even better. But! Friendship breaks abstraction boundaries, so you should only use it if there's a compelling argument for it.

- Friendship is neither symmetric nor transitive, so just because **`class A`** grants friendship to **`class B`** doesn't mean that **`A`** gets access to **`B`**'s private data. And if **`A`** grants friendship to **`B`**, and **`B`** grants friendship to **`C`**, **`A`** is not granting friendship to **`C`** through **`B`**.

More friends

The real **`string`** class also allows its objects to be compared to one another. We could add **`bool string::operator==(const string& other) const`** to **`string`**'s **`public`** section, but that would introduce asymmetrical support for comparison to C string constants. More often than not, I see the comparison operators supported as two-

argument predicate **friend** functions, not as **public** one-argument methods. (Again, there's a very good reason for going with **friend** here.)

Here's our own **string** class extended to support the six different relational operators:

```
class string {

   friend const string operator+(const string& first, const string& second);

   friend bool operator==(const string& first, const string& second);
   friend bool operator!=(const string& first, const string& second);
   friend bool operator<(const string& first, const string& second);
   friend bool operator<=(const string& first, const string& second);
   friend bool operator>(const string& first, const string& second);
   friend bool operator>=(const string& first, const string& second);

   public:
      string(const char *str = "");
      string(const string& original);
      const string& operator=(const string& rhs);
      ~string();

      char& operator[](int i) { return front[i]; }
      char operator[](int i) const { return front[i]; }
      const string& operator+=(const string& rhs);

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};
```

Here are the implementations of the first three (these would reside in **string.c**)

```
bool operator==(const string& first, const string& second)
{
   return strcmp(first.front, second.front) == 0;
}

bool operator!=(const string& first, const string& second)
{
   return strcmp(first.front, second.front) != 0;
}

bool operator<(const string& first, const string& second)
{
   return strcmp(first.front, second.front) < 0;
}
```

More Random Words

- You should only include operators that have a natural interpretation within the domain of the type you're defining. In the case of the **`string`** class, it makes sense to support **`+`**, **`[]`**, and **`+=`**, but it doesn't make sense to support **`%`**, **`/=`**, **`&&`**, and **`++`** just because you can. For most classes, you shouldn't overload any operators at all (except maybe **`operator=`**). For objects that need to be stored as keys within an STL **`map`** or **`set`**, you might implement **`==`** and **`<`**. For a few classes, you might implement a subset of all the operators (as we have with the **`string`** class), and for very few (like a **`fraction`**, **`complex`**, or **`bigint`** class), you might implement nearly all of them.

- Make sure the operators you **do** support are implemented in a way that user would expect. Don't be cute and implement operating **`operator->()`** to open a network connection and download the Web. The prevailing philosophy is to set your operators to "do as the primitives do", meaning **`-`** means minus, **`++`** means increment by 1, **`*`** means multiply, **`[]`** means array access, and **`<<`** means publish-to-stream.

- You can't invent new operators. You can only introduce added meaning to the ones that already exist. This is because the compiler's stream tokenizer can be (and probably is) written with the predefined token set of mind. g++ can't accept ## as a legitimate token just because you feel a need for it.

- You can't change the precedence rules governing order of evaluation. This is because the C++ lexer and parser have the precedence rules built in, and it would complicate the post-parsing implementation if precedence rules need to change on a type-by-type basis.

- You lose short-circuit evaluation if you overload **`&&`**, **`||`**, and/or **`!`**. I'm not sure why, but you do.

- You can't displace existing implementation of the built-in operators for the built-ins, so don't even think about redefining what it means to add two **`double`**s or to exclusive-or two **`bool`**s.

- Some operators can't be overloaded: **`.`** (field and method access within an aggregate type), **`::`** (the scope operator), **`?:`** (the ternary operator), and **`sizeof`**. Sorry about that. (You can overload the comma operator, though.)

The `bigint` class

For fun, I decided to implement a **`bigint`** class, just to see if I could do it, but also because I originally planned to use it as a vehicle for illustrating how all operators could be overloaded (including ones like **`++`**, **`--`**, and **`*=`**). Here's the sample program I wrote to unit test some of the **`bigint`** methods I'm including, and the test output I ultimately got once everything seemed to be working properly:

```
// bigint-test.cc
int main()
{
   cout << "Here are the first 90 factorials:" << endl;
   cout << "---------------------------------" << endl;
   for (bigint n = 0; n <= 90; n++) {
      cout << setw(2) << n << ".) ";
      bigint factorial = 1;
      for (bigint factor = 1; factor <= n; factor++) {
         factorial *= factor;
      }
      cout << factorial << endl;
   }
}

// output
Here are the first 50 factorials:
---------------------------------
 0.) 1
 1.) 1
 2.) 2
 3.) 6
 4.) 24
 5.) 120
 6.) 720
 7.) 5040
 8.) 40320
 9.) 362880
 10.) 3628800
 11.) 39916800
 12.) 479001600
 13.) 6227020800
 14.) 87178291200
 15.) 1307674368000
 16.) 20922789888000
 17.) 355687428096000
 18.) 6402373705728000
 19.) 121645100408832000
 20.) 2432902008176640000

 // snip ☺

 45.) 119622220865480194561963161495657715064383733760000000000
 46.) 5502622159812088949850305428800254892961651752960000000000
 47.) 258623241511168180642964355153611979969197632389120000000000
 48.) 12413915592536072670862289047373375038521486354677760000000000
 49.) 608281864034267560872252163321295376887552831379210240000000000
 50.) 30414093201713378043612608166064768844377641568960512000000000000
```

Here's the interface for the **bigint** class. The full implementation of **bigint.cc** is posted to the right of the Handout 03 link on at **http://cs107l.stanford.edu**. But if you look at the next page, I work through the implementations of **operator<<** and both versions of **operator++**.

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

class bigint {

   friend ostream& operator<<(ostream& os, const bigint& val);

   friend bool operator==(const bigint& val1, const bigint& val2);
   friend bool operator!=(const bigint& val1, const bigint& val2);
   friend bool operator<(const bigint& val1, const bigint& val2);
   friend bool operator<=(const bigint& val1, const bigint& val2);
   friend bool operator>(const bigint& val1, const bigint& val2);
   friend bool operator>=(const bigint& val1, const bigint& val2);

   friend const bigint operator+(const bigint& one, const bigint& two);
   friend const bigint operator-(const bigint& one, const bigint& two);
   friend const bigint operator*(const bigint& one, const bigint& two);

   public:
      bigint(int val = 0);
      bigint(const string& text);
      // compiler-synthesized copy constructor,
      // operator=, and destructor all do the right thing

      const bigint& operator+=(const bigint& rhs);
      const bigint& operator-=(const bigint& rhs);
      const bigint& operator*=(const bigint& rhs);

      const bigint& operator++();
      const bigint operator++(int unused);
      const bigint& operator--();
      const bigint operator--(int unused);

      const bigint& operator+() const;
      const bigint operator-() const;

   private:
      vector<char> digits;
      bool isNegative;

      void initFromText(string numericText);
      void trim();
      void addDigits(const bigint& rhs);
      void subtractDigits(const bigint& rhs);
      static int magnitudeDiff(const bigint& one, const bigint& two);
      const bigint singleDigitMultiply(int digit, int extraZeroes) const;
};
```

Feature Implementations

Each **bigint** object is backed by a **vector<char>**, where the characters are constrained to be digits. They're stored in reverse order (one digit at position 0, tens digits at position 1, and so forth) to help simplify the implementation of the arithmetic operations. The **trim** routine normalizes the representation and hacks off any extraneous leading zeroes.

When time comes to print the **bigint**, we rely on the overloaded **operator<<** function:

```
ostream& operator<<(ostream& os, const bigint& value)
{
   if (value.isNegative) os << "-";
   for (int i = value.digits.size() - 1; i >= 0; i--) {
      os << value.digits[i];
   }

   return os;
}
```

Whenever you see **cout << data**, what you're really seeing is either a call to **cout.operator<<(data)**, or a call to **operator<<(cout, data)**. The **ostream** class and its subclasses provide one-argument **operator<<** methods for all of the primitives, and library classes generally provide two-argument **operator<<** functions.

Whenever you see a chain of **<<** calls, as with **cout << data1 << data2 << data3**, what you're really seeing executed is something like:

```
cout.operator<<(data1).operator<<(data2).operator<<(data3);
```
or

```
operator<<(operator<<(operator<<(cout, data1), data2), data3);
```
or

```
operator<<(cout.operator<<(data1).operator<<(data2), data2),
```
or

some other mix of method and function calls. In order the integrate your new **operator<<** into the fold, you need to declare a two-argument function (usually a **friend** of your class, though not required) that takes an **ostream&** and a const reference to an instance of your new type, and returns a reference to the incoming **ostream&** so that cascaded calls to **<<** can be made. (Technically, you could go into the **ostream.h** files and add an **operator<<** method for your new type, but that's not a very scalable solution, and it would require that you recompile the standard libraries, and that's not going to happen. Your custom data type isn't **that** important.)

You can also implement **operator>>** if you'd like to read a **bigint** in from **cin** or some **ifstream**, but I'll leave you to figure out the details of that.

The implementation of **bigint::operator++()** is invoked every time you levy a prefix **++** against a **bigint**. The distinction here is that prefix **++** is supposed to return the incremented value. The implementation (assuming a fully operational **+=**) would be:

```
const bigint& bigint::operator++()
{
   *this += 1;
   return *this;
}
```

Remember, we're doing as the **int**s would do, so prefix **++** is supposed to increment the value held by the affected **bigint** and then return the new value. We're able to return ***this** by **const** reference, because the receiving object stores the result after the increment. If we're able to return a reference instead of a full object, we return a reference.

The functionality of postfix **++** is handled by another version of **operator++**—**bigint::operator++(int unused)**. The implementation, unlike the prefix version, is supposed to return the **old** value, not the new one. That means we have to clone the state of ***this** before we actually go through with the increment so we know what value to return. The implementation of postfix **++** looks like this:

```
const bigint bigint::operator++(int unused)
{
   bigint clone(*this);    // take snapshot of old value
   operator++();           // call other version to handle the increment
   return clone;           // return old value (via copy constructor)
}
```

At compile time, all prefix **++** calls are translated to **operator++()** calls, and all postfix **++** operations are translated to **operator++(0)** calls. Yes, the compiler silently passes in a dummy **0** to **operator++** when it wants the postfix version to be called instead. It's quite possibly the biggest hack in all of C++.