Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

1 Overloading the plus (+) operator

Consider the following definition of `class String`:

```
1  #include <cstring>
2  using namespace std;
3
4  class String
5  {
6  private:
7     char* str; // C−string containing the string appended by the null char
8     int n;     // Number of characters in str, not including the null char
9
10 public:
11    String() {n=0; str=0;}
12
13    String(const char* cstr)
14    {n=strlen(cstr); str=new char[n+1]; strcpy(str, cstr);}
15
16    String(const String& s)
17    {n=s.n; str=new char[n+1]; strcpy(str, s.str);}
18
19    ~String() {if(str) delete[] str;}
20
21    void Output() {cout<<str<<endl;}
22
23    String Add(String b) // Add this to b and return the result
24    {                    //  without changing this or b
25       String r;
26       r.n=n+b.n;  if(r.n==0) return r;
27       r.str=new char[r.n+1];
28       if(str) strcpy(r.str, str);
29       if(b.str) strcpy(r.str+n, b.str);
30       return r;
31    }
32 };
```

The class includes an empty constructor and a one-argument constructor which takes a C-string and converts it to an our user-defined `String` object. The class includes also a copy constructor which can be used explicitly, and is also used implicitly when passing a `String` object by value (such as in `Add()` function). The `Add()` function adds two strings and returns the result. The first added object is the one which called the function, the second added object is the only parameter to the function. Consider the following program which uses the previously defined class:

```
int main()
{
   String a("Hello"), b("-World");
   String c=a.Add(b);
   a.Output(); // Prints: Hello
   b.Output(); // Prints: -World
   c.Output(); // Prints: Hello-World
   return 0;
}
```

To avoid calling the copy constructor which increases efficiency, we may pass the argument to the `Add()` function by reference. Since that parameter is not going to be modified, we may add the `const` modifier to improve readability and also to improve reliability by asking the compiler to flag an error if the programmer mistakenly modifies the parameter within the `Add()` function:

```
String Add(const String& b);
```

Since the object which calls the `Add()` function is not going to be modified by it, we should add the `const` modifier at the end of the function declaration as follows:

```
String Add(const String& b) const;
```

Suppose we want to use the + operator instead of the `Add()` function to improve readability:

```
String a="Hello", b="-World";    // the = in declaration calls constructor
String c=a+b;     // Instead of: String c=a.Add(b);
```

Note that the operand before the + operator will be considered as the object who called the operator. The operand after the + operator will be passed as the only argument for the operator. This can be achieved by replacing the `Add()` function inside `class String` by:

```
class String
{ //  .....
   String operator + (const String& b) const
   {
      String r;  r.n=n+b.n;  if(r.n==0) return r;
      r.str=new char[r.n+1];  if(str) strcpy(r.str, str);
      if(b.str) strcpy(r.str+n, b.str);
      return r;
   }
};
```

Suppose we need to write the following code:

```
String a="Hello";  const char* b="-World";  String c=a+b;
```

Although the second argument type is not `String`, the above code will work because the compiler will be able to convert a `char*` type to a `String` type using the constructor we provided: `String(const char* cstr)`, and then the above definition of the `+` operator.

However, for efficiency, we may write another member function of class `String` that will be called if the first operand of the `+` operator is a `String` and the second operand is `char*`:

```
class String
{ // .....
   String operator + (const char* b) const
   {
      String r;  int nb=strlen(b);  r.n=n+nb;
      if(r.n==0) return r;  r.str=new char[r.n+1];
      if(str) strcpy(r.str, str);  if(b) strcpy(r.str+n, b);
      return r;
   }
};
```

Now, suppose we need to write the following code:

```
const char* a="Hello";   String b="-World";  String c=a+b;
```

One may expect that the above code will work as in the previous case. But, the code will not work because the object which called the operator (the left operand) is of type `char*`. So, the compiler will search for an overloaded function in class `char*` that takes a `String` parameter, and it will fail. The compiler will not attempt to search for an overloaded function in `class String` because the calling object (left operand) is not of type `String`.

Where to write a function overloading the `+` operator that takes a `char*` as its left operand? It should be written in `class char*` to which we do not have access. Even if we have access to it, it is not a good idea to modify an old tested class for such purpose. The solution is to make such function global, and make it `friend` of `class String` to access its private members easily:

```
class String
{ // .....
   friend String operator + (const char*, const String&);
};

String operator + (const char* a, const String& b)
{
   String r;  int na=strlen(a);  r.n=na+b.n;
   if(r.n==0) return r;  r.str=new char[r.n+1];
   if(a) strcpy(r.str, a);  if(b.str) strcpy(r.str+na, b.str);
   return r;
}
```

2 Overloading the assignment (=) operator

Consider the following code:

```
1  int main()
2  {
3     String a="Hello";    // Same as: String a("Hello");
4     String b=a;          // Same as: String b(a);
5     a.Output(); // Prints: Hello
6     b.Output(); // Prints: Hello
7     return 0;
8  }
```

The above code works because it actually calls the constructor, not the assignment = operator.

Consider the following code which attempts to use the assignment = operator:

```
1  String a="Hello";    // Same as: String a("Hello");
2  String b;   b=a;       // Calls the empty constructor then the assignment operator
```

The above code will compile and run, but it will not work as expected. Since we did not overload a copy assignment = operator in `class String`, the compiler will provide a default copy assignment operator as follows:

```
1  class String
2  { //  .....
3     String& operator = (const String& b)
4     {
5        n=b.n;   str=b.str;
6        return *this;
7     }
8  };
```

The problem of the above code is that both strings now share the same area of allocated memory, because `a.str=b.str`. So when `a` or `b` is destroyed, the destructor will de-allocate the memory used by the other object as well. The correct way is to overload the copy assignment operator so as not to let the compiler provide the default one:

```
1  class String
2  { //  .....
3     String& operator = (const String& b)
4     {
5        n=b.n;   if(str) delete[] str;   str=0;
6        if(!b.str) return *this;
7        str=new char[n+1];   strcpy(str, b.str);
8        return *this;
9     }
10 };
```

For a similar reason, we overloaded the copy constructor `String(const String& s)` in the first program of this lecture so as not to let the compiler provide the problematic default copy constructor which looks like that:

```
class String
{ // .....
   String(const String& b) : n(b.n), str(b.str) {}
};
```

For other classes like `class Fraction`, the compiler-provided default copy constructor and default copy assignment operator are satisfactory.

But why the assignment operator returns `*this`? The importance of the return value is to make a statement like this work: `x=y=b;` which invokes `y=b` first because of the right associativity of the assignment operator, then the return value is assigned to `x`. Since we can just modify the calling object `y` and return it, we used `String&` return type to avoid creating and returning a new object which is less efficient. We are able to return it by reference because we are sure that its lifetime continues after the assignment operator is called.

If our implementation does not have a return value (`void` instead of `String&`), the program will work for `y=b;` but it will not compile for `x=y=b;`. Even if this behavior is well-described in your documentation, users of your class will be confused because `x=y=b;` work well for basic data types such as `int` and `double`. It is good practice to overload operators such that they behave similarly to their behavior with basic data types.

3 Overloading the += operator

In this section we define the binary `+=` operator so that it appends the right operand `String` object to the left operand `String` object, then returns the modified left operand `String` object. The returned object allows statements like this to work: `x=y+=b;`.

```
class String
{ // .....
   String& operator += (const String& b)
   {
      if(b.n==0) return *this;
      int new_n=n+b.n;
      char* new_str=new char[new_n+1];
      if(str) strcpy(new_str, str);
      if(b.str) strcpy(new_str+n, b.str);
      n=new_n;
      if(str) delete[] str;
      str=new_str;
      return *this;
   }
};
```

4 Overloading the >> and << operators

Suppose we need the following code to work for `class String` objects as it works for basic data types:

```
String a, b;
cin>>a>>b;    // Assume the user inputs: Hello World
cout<<a<<"-"<<b;    // Prints: Hello-World
```

`cin` is a global object of type `istream`. That object is somehow associated with the standard input stream in the C++ library. Since the operator `>>` is left associative, the statement `cin>>a>>b;` is executed by first executing `cin>>a`, which inputs `a` from user then returns the same `cin` object again so as to let the `cin>>b` statement executed.

In the above code, the `>>` operator takes an `istream` left operand and `String` right operand. Therefore, its overloading function can not be a member of `class String` (we encountered this situation before when we tried to overload the `+` operator such that it takes a `const char*` left operand and `String` right operand). We can only overload the `>>` by adding a member function in `class istream` (which is not possible because it is a library class), or by defining a global function and making it a `friend` of `class String` to easily access its members.

```
class String
{ //  .....
   friend istream& operator >> (istream&, String&);
};

istream& operator >> (istream& in, String& s)
{
   char buf[200];  buf[0]=0;  in>>buf;
   if(s.str) delete[] s.str;
   s.n=strlen(buf);  s.str=0;  if(s.n==0) return in;
   s.str=new char[s.n+1];  strcpy(s.str, buf);
   return in;
}
```

Similarly, the `<<` operator can be overloaded as follows (Note that `cout` is a global object of type `ostream` which is associated with the standard output stream in the C++ library):

```
class String
{ //  .....
   friend ostream& operator << (ostream&, const String&);
};

ostream& operator << (ostream& out, const String& s)
{
   out<<s.str;  return out;
}
```

5 Overloading the prefix and postfix ++ operators

Suppose we need to define the unary prefix and postfix `++` operators to work on `class Fraction` objects similarly to other numerical data types:

```
Fraction f(3, 5);   // f=3/5

Fraction g = ++f;
f.Output();  g.Output();   // Prints:  8/5 8/5

Fraction h = f++;
f.Output();  h.Output();   // Prints:  13/5 8/5
```

Since the `++` operator is unary (Note that all operators defined in previous sections are binary), it takes only one operand which is the calling object. Therefore, the overloaded functions should not take any parameters because they access the calling operand using `this`. We should overload the `++` operator as follows:

```
class Fraction
{ //  .....
   Fraction& operator ++ ()     // Prefix ++ operator
   {
      num += den;              // Add by one
      return *this;            // Return the modified value
   }

   Fraction operator ++ (int)      // Postfix ++ operator
   {
      Fraction f = *this;          // Save the current value
      num += den;                  // Add by one
      return f;                    // Return the old value
   }
};
```

Note that the postfix `++` operator does not take an integer parameter as it looks. The `int` keyword here is just a way to tell the compiler that we are overloading the postfix operator, not the prefix one.

6 Overloading the brackets [] and parentheses () operators

Suppose we need the `[]` operator work for `class String` objects our as it works for arrays:

```
String a = "Hello";
cout << a[1] << endl;   // Prints: e
a[2] = 'x';
a.Output();   // Prints: Hexlo
```

We should overload the `[]` operator as follows:

```
class String
{ //  .....
   char& operator [] (int i) {return str[i];}
};
```

The `[]` operator takes one parameter, which is the integer inside the brackets. The return value should be a `char&` so as to be able to modify that specific character inside the string, such as the statement `a[2]='x';`.

Overloading the `()` operator is similar but it can take any number of parameters inside:

```
class IntegerMatrix
{ //  .....
   int** mat;
   int& operator () (int i, int j) {return mat[i][j];}
};
```

The `()` operator can be unary (if no parameters inside parentheses), binary, or more (because it can have any number of parameters inside parentheses, including zero). Overloading the `()` is interesting because it makes an object looks like a function.

7 Type conversion

Consider the following code:

```
String a, b, c;
a = "Hello";   // Implicit conversion
b = (String) "World";   // Explicit conversion
c = static_cast<String>("Prog");   // Explicit conversion
a.Output(); b.Output(); c.Output();   // Prints: HelloWorldProg
```

For the first assignment, because the left operand type is `String`, the compiler searches in `class String` for an overloading of the assignment operator which takes a `const char*` as parameter (the type of the right operand). Since such function does not exist, the compiler will keep searching for a function in `class String` which may work if the compiler performed the necessary conversions to match its parameter types. The compiler figures out that it can apply the constructor `String(const char* cstr)` to convert the right operand from `const char*` to `class String` then to invoke the overloaded assignment operator.

For the second and third assignments, which are equivalent, the compiler will immediately search for a constructor that takes a `const char*` as parameter so as to convert the right operand to `String` as explicitly requested, then it will invoke the overloaded assignment operator.

Therefore, to convert from any data type `X` to type `String`, a one-argument constructor is needed in `class String` whose parameter type is `X`. Now, how to convert from a `String` type to another data type? We can define a constructor that takes a `String` parameter in the class of that data type if possible, or we can add a member function in `class String` as follows:

```
class String
{ //  .....
   operator const char*() const {return str;}
};
```

Which will make the following code works:

```
String a = "Hello";
const char* x = a;   // Implicit conversion
const char* y = (const char*) a;   // Explicit conversion
const char* z = static_cast<const char*>(a);   // Explicit conversion
cout<<x<<" "<<y<<" "<<z<<endl;   // Prints: Hello Hello Hello
```