C++’s Rule of 3 and Rule of 5

Rule of 3: If a class needs to define any one of these 3 methods, then likely the class should define all 3 of these methods

- Destructor
- Copy constructor
- (Copy) Assignment operator

Rule of 5 (as of C++ 11): If a class needs to define any one of these 5 methods, then likely the class should define all 5 of these methods

- Destructor
- Copy constructor
- (Copy) Assignment operator
- Move constructor
- Move assignment operator

Example:

```
#include <iostream>
#include <cstring>
using namespace std;

class MyString {
        char* data;
public:
        MyString( ) {                                   // default constructor
                cout << "Default constructor" << endl;
                data = new char[6];
                strcpy (data, "Hello");
        }
        MyString (const char *str) {                    // another constructor
                cout << "Another constructor: " << str << endl;
                data = new char[strlen (str) + 1];
                strcpy (data, str);
        }
        ~MyString( ) {                                  // destructor
                cout << "Destructor: ";
                if (data==nullptr) cout << "nullptr" << endl;
                else cout << data << endl;
                delete[ ] data;
        }
        MyString (const MyString& other) {              // copy constructor
                cout << "Copy constructor: " << other << endl;
                data = new char[strlen (other.data) + 1];
                strcpy (data, other.data);
        }
```

```
    MyString& operator= (const MyString& other) {
                                        // (copy) assignment operator
        cout << "Copy assignment operator: " << other << endl;
        MyString temp(other);           // use copy constructor
        *this = move(temp);             // force move semantics,
                                    // use move assignment operator
        return *this;
    }
    MyString (MyString&& other) {       // move constructor,
                                        // as of C++ 11
        cout << "Move constructor: " << other << endl;
        data = other.data;
        other.data = nullptr;
    }
    MyString& operator= (MyString&& other) {
                                        // move assignment operator,
                                        // as of C++ 11
        cout << "Move assignment operator: " << other << endl;
        delete[ ] data;
        data = other.data;
        other.data = nullptr;
        return *this;
    }
private:
    friend ostream& operator<< (ostream& os, const MyString& str) {
        os << str.data;
        return os;
    }
```

```
        friend MyString& operator+ (const MyString& s,
                                    const MyString& t) {
                int len = strlen(s.data) + strlen(t.data) + 1;
                char *str = new char[len];
                strcpy (str, s.data);
                strcat (str, t.data);
                MyString *temp = new MyString(str);
                return *temp;
        }
};

int main( ) {
        MyString s;
        cout << "s = " << s << endl << endl;

        MyString t (",World!");
        cout << "t = " << t << endl << endl;

        MyString u (s);
        cout << "u = " << u << endl << endl;

        u = t;
        cout << "u = " << u << endl << endl;

        MyString v (move(s + t));
        cout << "v = " << v << endl << endl;

        u = move(s + t);
        cout << "u = " << u << endl << endl;
        return 0;
}
```

Output:

```
Default constructor
s = Hello

Another constructor: ,World!
t = ,World!

Copy constructor: Hello
u = Hello

Copy assignment operator: ,World!
Copy constructor: ,World!
Move assignment operator: ,World!
Destructor: nullptr
u = ,World!

Another constructor: Hello,World!
Move constructor: Hello,World!
v = Hello,World!

Another constructor: Hello,World!
Move assignment operator: Hello,World!
u = Hello,World!

Destructor: Hello,World!
Destructor: Hello,World!
Destructor: ,World!
Destructor: Hello
```

Explanation of Move semantics:

First consider the declaration MyString u (s);

- Here the value of s can change, so we must make a copy of s to place into u
- That is, s is an L-value (can appear on the left side of an assignment statement)
- L-values retain their value into subsequent statements

Now consider the declaration MyString v (s + t);

- Here s + t is an expression, so its value cannot change, and so we can move its value into v rather than making a copy
- That is, s + t is an R-value (can only appear on the right side of an assignment statement)
- R-values denote temporary values that will be destroyed at the end of the current statement