Operator Overloading Part 2

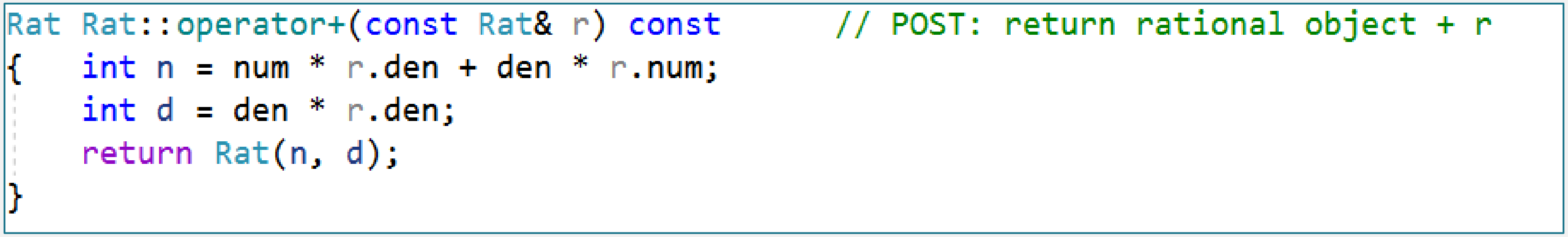
Writing a Rational Number Class

Operators << and >>

Overloading = Operator

Operators as Class Members

Typically, an operator is a class member



```
Rat Rat::operator+(const Rat& r) const      // POST: return rational object + r
{   int n = num * r.den + den * r.num;
    int d = den * r.den;
    return Rat(n, d);
}
```

When used, the left operand makes the function call and the right operand is passed in

```
Rat r1 (1, 2);
Rat r2 (3, 4);
Rat r3;
r3 = r1+r2;       // r1 makes the call
                  // r2 is passed to parameter r
```

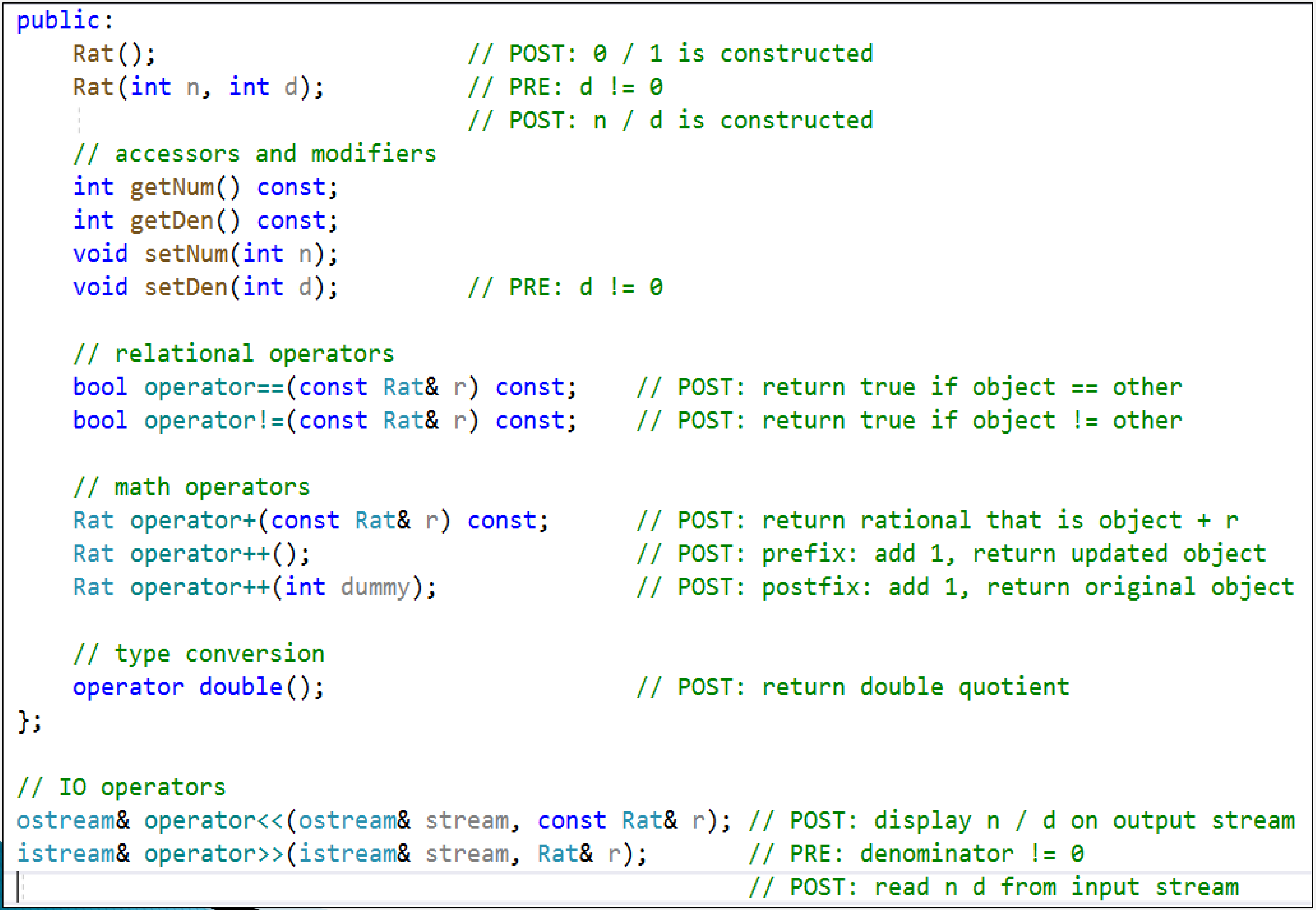
IO Operators

IO operators << and >> cannot be implemented effectively as class members

```
cout << 25;
```

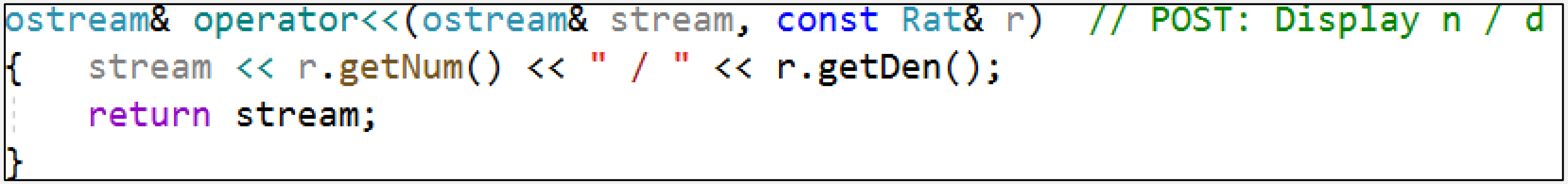
The left operand is the stream
The right operand is the data

```
Rat r1 (1, 2);
cout << r1;
```



```
public:
    Rat();                      // POST: 0 / 1 is constructed
    Rat(int n, int d);          // PRE: d != 0
                                // POST: n / d is constructed
    // accessors and modifiers
    int getNum() const;
    int getDen() const;
    void setNum(int n);
    void setDen(int d);         // PRE: d != 0

    // relational operators
    bool operator==(const Rat& r) const;    // POST: return true if object == other
    bool operator!=(const Rat& r) const;    // POST: return true if object != other

    // math operators
    Rat operator+(const Rat& r) const;      // POST: return rational that is object + r
    Rat operator++();                       // POST: prefix: add 1, return updated object
    Rat operator++(int dummy);              // POST: postfix: add 1, return original object

    // type conversion
    operator double();                      // POST: return double quotient
};

// IO operators
ostream& operator<<(ostream& stream, const Rat& r); // POST: display n / d on output stream
istream& operator>>(istream& stream, Rat& r);       // PRE: denominator != 0
                                                    // POST: read n d from input stream
```

<< Operator



```
ostream& operator<<(ostream& stream, const Rat& r)  // POST: Display n / d
{   stream << r.getNum() << " / " << r.getDen();
    return stream;
}
```

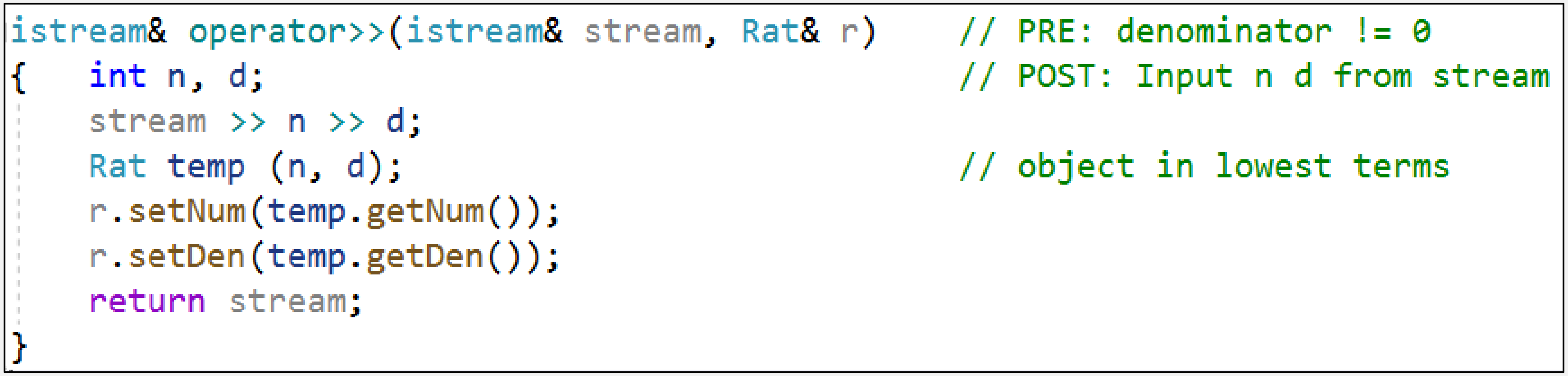
Parameter ostream can be passed any type of output stream
Streams are always passed by reference as there is only one stream object

The return type is ostream& for associativity

Rat r1 (1, 2);
Rat r2 (3, 4);
cout << r1 << r2;

Use public accessor methods as << is not a class member

>> Operator

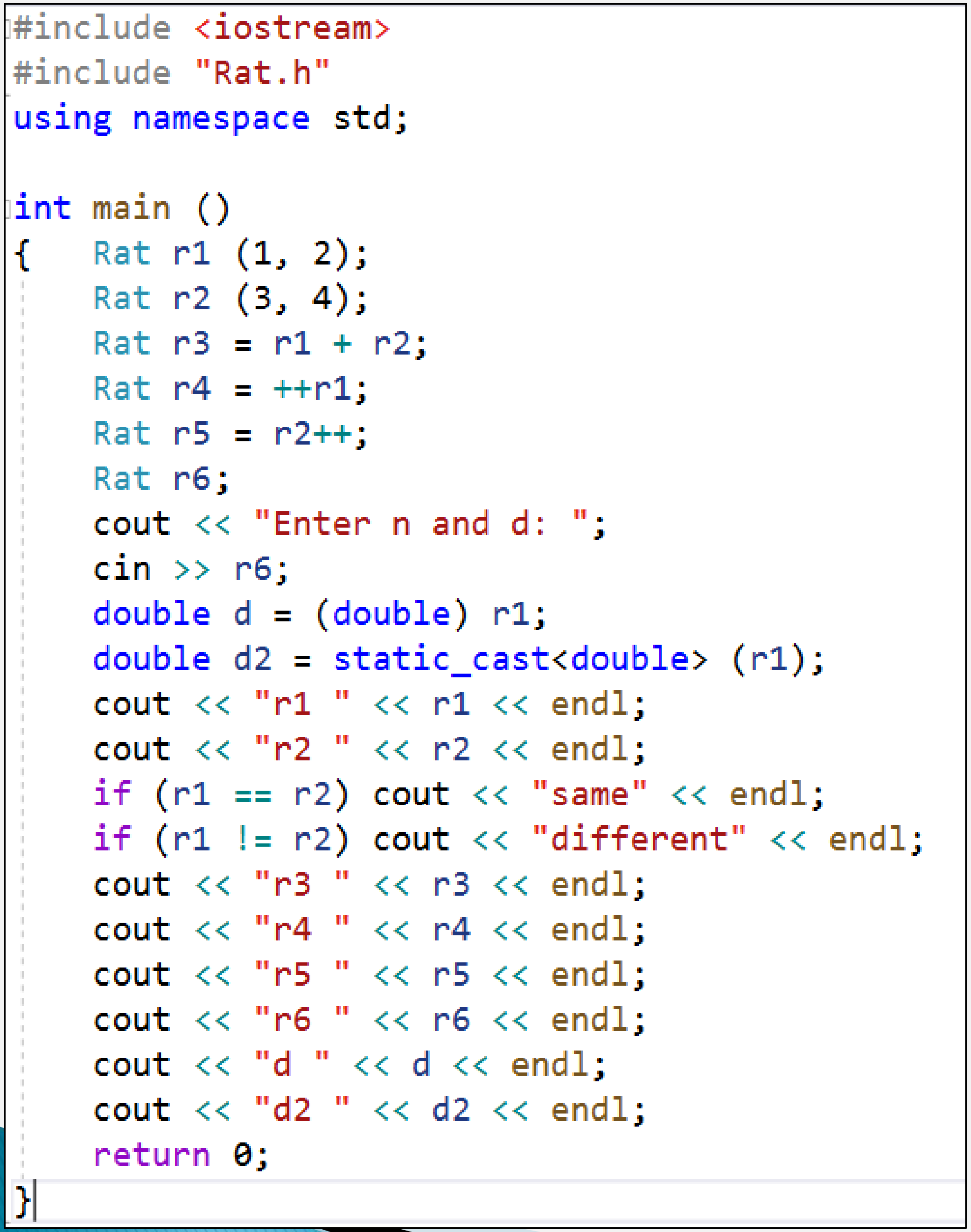


```
istream& operator>>(istream& stream, Rat& r)    // PRE: denominator != 0
{   int n, d;                                   // POST: Input n d from stream
    stream >> n >> d;
    Rat temp (n, d);                            // object in lowest terms
    r.setNum(temp.getNum());
    r.setDen(temp.getDen());
    return stream;
}
```

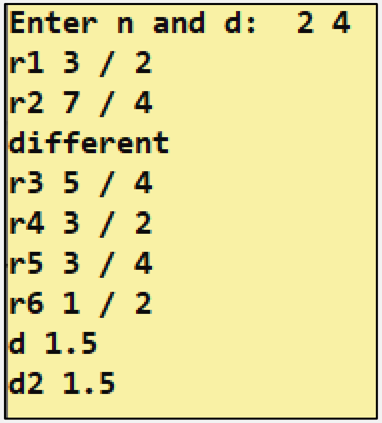
Parameter istream can be passed any type of input stream
Parameter r is passed by reference as it will be written
The return type is istream& for associativity

Rat r1;
cin >> r1;

Use public accessor methods as >> is not a class member



```cpp
#include <iostream>
#include "Rat.h"
using namespace std;

int main ()
{   Rat r1 (1, 2);
    Rat r2 (3, 4);
    Rat r3 = r1 + r2;
    Rat r4 = ++r1;
    Rat r5 = r2++;
    Rat r6;
    cout << "Enter n and d: ";
    cin >> r6;
    double d = (double) r1;
    double d2 = static_cast<double> (r1);
    cout << "r1 " << r1 << endl;
    cout << "r2 " << r2 << endl;
    if (r1 == r2) cout << "same" << endl;
    if (r1 != r2) cout << "different" << endl;
    cout << "r3 " << r3 << endl;
    cout << "r4 " << r4 << endl;
    cout << "r5 " << r5 << endl;
    cout << "r6 " << r6 << endl;
    cout << "d " << d << endl;
    cout << "d2 " << d2 << endl;
    return 0;
}
```



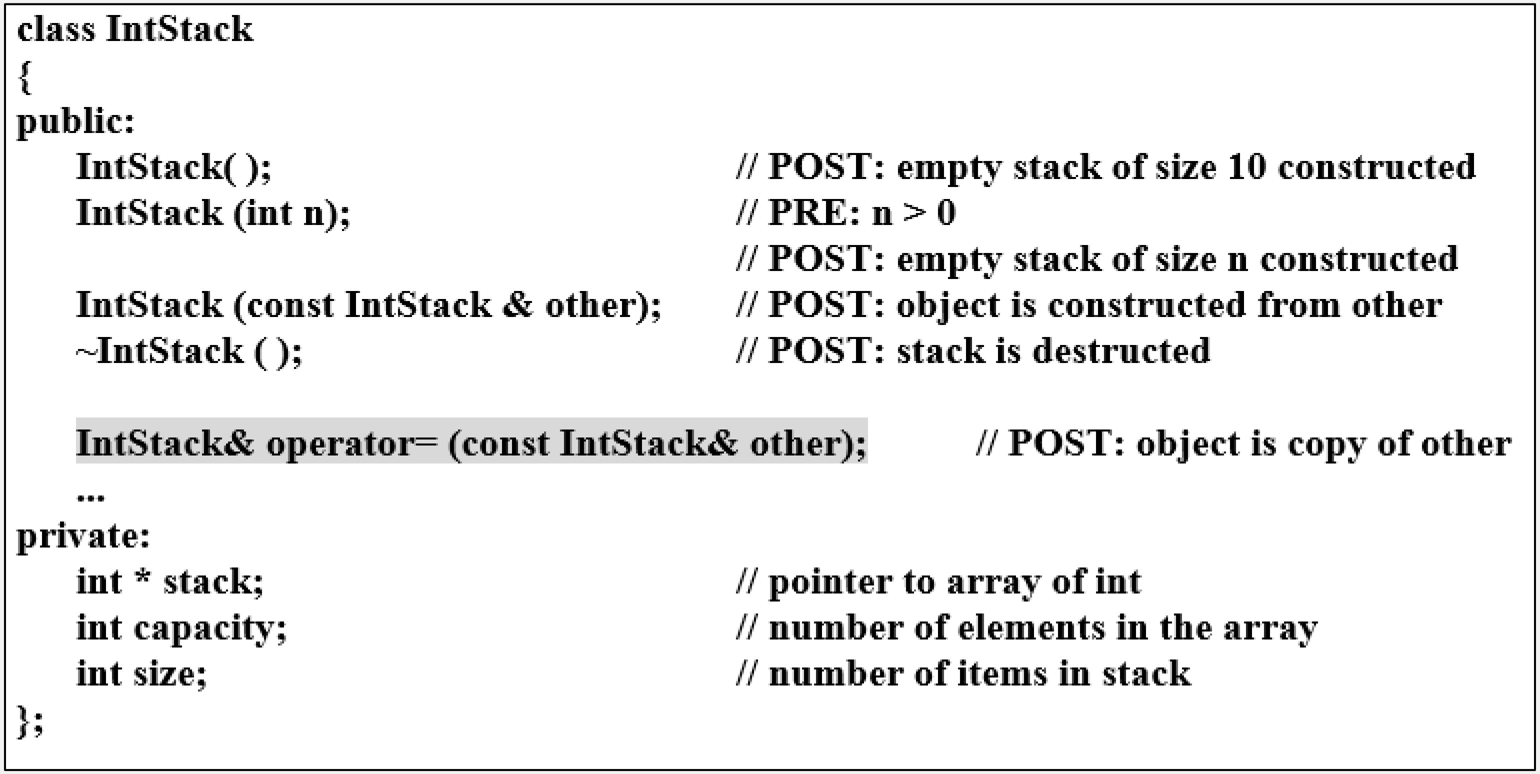
```
Enter n and d:  2 4
r1 3 / 2
r2 7 / 4
different
r3 5 / 4
r4 3 / 2
r5 3 / 4
r6 1 / 2
d 1.5
d2 1.5
```

Classes and Dynamic Memory

Classes that use dynamic memory include the functions:

- destructor
- copy constructor
- operator=

Stack class



```
class IntStack
{
public:
     IntStack( );                              // POST: empty stack of size 10 constructed
     IntStack (int n);                         // PRE: n > 0
                                               // POST: empty stack of size n constructed
     IntStack (const IntStack & other);        // POST: object is constructed from other
     ~IntStack ( );                            // POST: stack is destructed

     IntStack& operator= (const IntStack& other);          // POST: object is copy of other
     ...
private:
     int * stack;                              // pointer to array of int
     int capacity;                             // number of elements in the array
     int size;                                 // number of items in stack
};
```

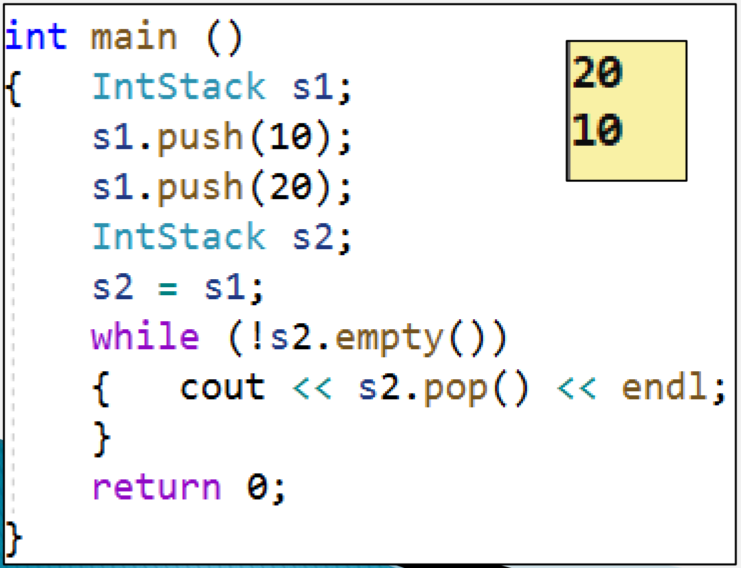
```
IntStack s1;
s1.push (25);
IntStack s2;
s2 = s1;
```

= is called on an existing object

Operator =

```
IntStack& IntStack::operator= (const IntStack& other)    // POST: object is a copy of other
{
        if (&other == this) return *this;        // handle odd case of self-assignment
        delete [ ] stack;                         // delete current dynamic array
        capacity = other.capacity;
        size = other.size;
        stack = new int [capacity];               // create new array
        for (int k = 0; k < size; k++)            // copy elements of other into new array
            stack[k] = other.stack[k];
        return *this;                             // return reference to object
}
```

s1 = s1;



```
int main ()
{    IntStack s1;
     s1.push(10);
     s1.push(20);
     IntStack s2;
     s2 = s1;
     while (!s2.empty())
     {    cout << s2.pop() << endl;
     }
     return 0;
}
```

```
20
10
```

Support chaining:
s3 = s2 = s1;