- ***Syntactic sugar***: Syntax in a programming language that makes something easier to express.

Example of syntactic sugar

```
int x = 1, y = 2;
int z = x + y;
```

```
int x = 1, y = 2;
int z = add(x, y)
```

Many operators are syntactic sugar because usually they are unnecessary in the language; we could get by with just functions.

Example of syntactic sugar

```
vector<int> vec(3);
vec[0] = 100;
cout << vec[0];
```

```
vector<int> vec(3);
vec.set(0, 100);
cout << vec.get(0);
```

Many operators are syntactic sugar because usually they are unnecessary in the language; we could get by with just functions.

Operator overloading

- ***Function overloading***: Allowing different functions with the same name, distinguished by argument number or data type(s).
- ***Operator overloading***: Adding new meanings for operators when used with different data types.

As simple as defining a function

- Define a function called:

  operator+ operator- operator* operator/
  operator+= operator< operator++ operator==

- Number of arguments is determined by the operator name.
  - i.e., operator+ always takes two arguments.
- Return type can be anything you want.

Overloading +

```
vector<int> vec1, vec2, vec3;
vec1.push_back(1);
vec1.push_back(2);
vec2.push_back(10);
vec2.push_back(20);
vec3 = vec1 + vec2;
```

Overloading +

```
vector<int> vec1, vec2, vec3;
vec1.push_back(1);
vec1.push_back(2);
vec2.push_back(10);
vec2.push_back(20);
vec3 = vec1 + vec2;
cout << vec3;
```

Overload these operators

```
vector<int> vec, vec2;
vec += 1;  // overload += so it does push_back
vec += 2;
vec += 1;
vec += 3;
vec2 = vec – 1;  // vec2 is now [2, 3]
                 // overload minus so it removes all
                 // all instances of an item from a vector
```

Overloading with classes

```
class rational {
  public: …
  private:
    int num, den;
};

rational operator* (const rational & a, const rational & b)
{
  rational ans;
  ans.num = a.num * b.num;
  ans.den = a.den * b.den;
  return ans;
}
```

Solution 1

```
class rational {
  public:
    rational operator*(const rational & b);
  private:
    int num, den;
};

rational rational::operator* (const rational & b)
{
  rational ans;
  ans.num = num * b.num;
  ans.den = den * b.den;
  return ans;
}
```

Solution 2

```
class rational {
  public:
    friend rational operator* (const rational & a, const rational & b)
  private:
    int num, den;
};

rational operator* (const rational & a, const rational & b)
{
  rational ans;
  ans.num = a.num * b.num;
  ans.den = a.den * b.den;
  return ans;
}
```

In your rational class

- Overload << (will need to be a friend)
- Overload + (put inside class)
- Overload < (put inside class)