Operator Overloading Example
class Complex

We wish to allow the following code:

```
Complex c1(5,4); // c1 is now  5 + i4
Complex c2(-5,4); // c2 is now  -5 + i4

c2++; // c2 is now  -4 + i4
++c2; // c2 is now  -4 + i5

c2 = c1 + c2; // c2 is now  5+i4-4+i5 = 1 + i9
c1 = c1 – c2; // c1 is now  5+i4-1-i9 = 4 – i5

c1 = -c1;  // c1 is now -4 + i5
```

```
#ifndef COMPLEX_H_
#define COMPLEX_H_

#include <iostream>
using std::ostream;

class Complex {
  double real;
  double imag;

public:
  Complex(double r, double i);

  const Complex operator+(const Complex &c) const;
  const Complex operator-(const Complex &c) const;

  Complex &operator+=(const Complex &c);
  Complex &operator-=(const Complex &c);

  Complex &operator++();
  Complex &operator++(int dummy);

  const Complex operator-() const;

  void print(ostream &out) const;
  void read(istream &in);
}

ostream &operator<<(ostream &out,
                    const Complex &c);
istream &operator>>(istream &in,
                    Complex &c);

#endif
```

```
Complex::Complex(double r, double i)
   : real(r), imag(i)
{}

const Complex Complex::operator+(const Complex &c) const
{
  Complex tmp = *this;
  tmp += c;
  return tmp;
}

const Complex Complex::operator-(const Complex &c) const
{
  Complex tmp = *this;
  tmp -= c;
  return tmp;
}

Complex &Complex::operator+=(const Complex &c)
{
  real += c.real; // this += is of double..
  imag += c.imag;
  return *this;
}

Complex &Complex::operator-=(const Complex &c)
{
  real -= c.real;
  imag -= c.imag;
  return *this;
}

const Complex Complex::operator-() const
{
  return Complex(-real, -imag);
}
```

```
Complex &Complex::operator++()
{
  imag++;
  return *this;
}

Complex &Complex::operator++(int dummy)
{
  real++;
  return *this;
}


void Complex::print(ostream &out) const
{
  out << real;
  if(imag==0) return;
  if(imag>=0) out << "+";
  out << imag << "i";
}

ostream &operator<<(ostream &out, const Complex &c)
{
  c.print(out);
  return out;
}


void Complex::read(istream &in)
{
  in >> real;
  in >> imag;
}

istream &operator>>(istream &in, Complex &c)
{
  c.read(in);
  return in;
}
```

Same as before, but using conversion from double to Complex, by c'tor, and using it to allow:

c1+c2	complex + complex
c1+5.0	complex + double
5.0+c1	double + complex

```
class Complex {
  double real;
  double imag;

public:
  Complex(double r=0.0, double i=0.0);

  Complex &operator+=(const Complex &c);
  Complex &operator-=(const Complex &c);

  Complex &operator++();
  Complex &operator++(int dummy);

  const Complex operator-() const;
}

const Complex operator+(const Complex &c1, const Complex &c2);

const Complex operator-(const Complex &c1, const Complex &c2);

ostream &operator<<(ostream &out, const Complex &c);

istream &operator>>(istream &in, Complex &c);
```

```
const Complex operator+(const Complex &c1, const Complex &c2)
{
  Complex tmp = c1;
  tmp += c2;
  return tmp;
}

const Complex operator-(const Complex &c1, const Complex &c2)
{
  Complex tmp = c1;
  tmp -= c2;
  return tmp;
}
```

Some common guidelines:

Say we write class C, then:

1) All parameters of type C will be passed as **`const &`**

2) If an operator changes the object it is working on:
   - It returns **`C&`**
   - Is not **`const`** (because it does change)
   - Returns **`*this`**

3) If an operator doesn't change the object it is working on:
   - It returns **`const C`**
   - It is declared **`const`**
   - It declares some temporary variable, manipulates it and returns it

4) Comparison operators return **`bool`**, and are always **`const`**

5) Binary operators whose left argument is not of type **`C`**, must be declared as a global function. Never make this global function friend, instead, define a public method and have this global function call it

Example:

```
C &C::operator+=(const C &c)
C &C::operator=(const C &c);

const C C::operator*(const C &c) const;
const C C::operator-(const C &c) const;

bool C::operator==(const C &c) const;

ostream &operator<<(ostream &out, const C &c)
{
  c.print(out);
  return out;
}

void C::print(ostream &out)
{
  …
}
```