OPERATOR OVERLOADING

OPERATOR OVERLOADING

IS A VERY POWERFUL - AND UNUSUAL
- FEATURE OF THE C++ LANGUAGE

IT ALLOWS THE PROGRAMMER TO
EXTEND COMMON OPERATORS TO
USER-DEFINED CLASSES

BTW, JAVA DOES NOT SUPPORT OPERATOR
OVERLOADING ALTHOUGH C# DOES.

OPERATOR OVERLOADING

ALLOWS THE PROGRAMMER TO EXTEND COMMON OPERATORS TO USER-DEFINED CLASSES

WHEN YOU OVERLOAD OPERATORS, YOU SHOULD BE CAREFUL TO MIRROR THE WAY THOSE OPERATORS WORK FOR BASIC TYPES

WHEN YOU OVERLOAD OPERATORS, YOU SHOULD BE CAREFUL TO MIRROR THE WAY THOSE OPERATORS WORK FOR BASIC TYPES

THIS IS IMPORTANT: THE USE OF THE OPERATOR NEEDS TO BE CONSISTENT ACROSS TYPES..

BUT IT DOES MAKE OPERATOR OVERLOADING MORE COMPLICATED THAN IT MIGHT SEEM

INTERNAL OR EXTERNAL OPERATORS?

ANOTHER ISSUE TO CONSIDER WITH OPERATORS IS WHETHER THEY SHOULD BE **MEMBER FUNCTIONS, OR GLOBAL (USUALLY FRIENDS)***

*NOT ALL EXTERNAL OPERATORS NECESSARILY HAVE TO BE FRIENDS, BUT IT IS OFTEN THE CASE

SOME OPERATORS HAVE TO BE EXTERNAL

SOME OPERATORS HAVE TO BE EXTERNAL

WHY? BECAUSE AN INTERNAL OPERATOR WILL ONLY WORK IF ONE OF THE OPERANDS IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber::operator+=(*(a->this),b);
```

SOME OPERATORS HAVE TO BE EXTERNAL

WHY? BECAUSE AN INTERNAL OPERATOR WILL ONLY WORK IF ONE OF THE OPERANDS IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

WILL ONLY WORK IF **a** IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber::operator+=(*(a->this),b);
```

SO THIS CODE WILL NOT WORK WITH AN INTERNAL OPERATOR

```
double a(3,5);
ComplexNumber b(7,6);
a += b;
```

BECAUSE `a` IS A BUILT-IN TYPE, NOT AN OBJECT

```
double a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
double a(3,5);
ComplexNumber b(7,6);
double::operator+=(*(a->this),b);
```

SO THIS CODE WILL NOT WORK WITH AN INTERNAL OPERATOR

```
double a(3,5);
ComplexNumber b(7,6);
a += b;
```

BECAUSE a IS A BUILT-IN TYPE, NOT AN OBJECT

```
double a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
double a(3,5);
ComplexNumber b(7,6);
double::operator+=(*(a->this),b);
```

THIS IS WHY SOME OPERATORS, SUCH AS << AND >> HAVE TO BE EXTERNAL

INTERNAL OR EXTERNAL OPERATORS?

SOME OPERATORS HAVE TO BE EXTERNAL

<< AND >> HAVE TO BE EXTERNAL

SOME OPERATORS HAVE TO BE INTERNAL

`operator=` ASSIGNMENT OPERATOR HAS TO BE INTERNAL
(ELSE THE DEFAULT C++ VERSION WILL STILL BE CALLED)

`(), [], ->` ALSO HAVE TO BE INTERNAL

SOME OPERATORS HAVE TO BE EXTERNAL

<< AND >> HAVE TO BE EXTERNAL

SOME OPERATORS HAVE TO BE INTERNAL

`operator=` ASSIGNMENT OPERATOR HAS TO BE INTERNAL (ELSE THE DEFAULT C++ VERSION WILL STILL BE CALLED)

`(), [], ->` ALSO HAVE TO BE INTERNAL

SOME OPERATORS CAN NOT BE OVERLOADED AT ALL

`::, ? :, .` CAN NOT BE OVERLOADED

WITH OTHER OPERATORS, INTERNAL OR EXTERNAL IS YOUR CHOICE

INTERNAL OR EXTERNAL OPERATORS?

IF YOU IMPLEMENT AN OPERATOR AS INTERNAL, ITS SIGNATURE WILL LOOK DIFFERENT THAN IF IT WERE EXTERNAL

INTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& ComplexNumber::operator++()
```

EXTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& operator++(ComplexNumber& )
```

INTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& ComplexNumber::operator++()
```

EXTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& operator++(ComplexNumber& )
```

THE INTERNAL OPERATOR HAS 1 LESS PARAMETER, BECAUSE THE RECEIVER IS IMPLICITLY PASSED IN VIA THE `this` POINTER

INTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& ComplexNumber::operator++()
```

EXTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& operator++(ComplexNumber& )
```

THE INTERNAL OPERATOR HAS 1 LESS PARAMETER, BECAUSE THE RECEIVER IS IMPLICITLY PASSED IN VIA THE `this` POINTER

ALSO - UNLIKE THE EXTERNAL OPERATOR, THE INTERNAL OPERATOR WILL ALWAYS BE REFERRED TO WITH THE SCOPE RESOLUTION OPERATOR ::

INTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& ComplexNumber::operator++()
```

EXTERNAL PRE-INCREMENT OPERATOR

```
ComplexNumber& operator++(ComplexNumber& )
```

THE INTERNAL OPERATOR HAS 1 LESS PARAMETER, BECAUSE THE RECEIVER IS IMPLICITLY PASSED IN VIA THE `this` POINTER

ALSO - UNLIKE THE EXTERNAL OPERATOR, THE INTERNAL OPERATOR WILL ALWAYS BE REFERRED TO WITH THE SCOPE RESOLUTION OPERATOR ::

BUT THE RETURN TYPES ARE THE SAME

OH, AND BTW -

OVERLOADING OPERATORS DOES NOT CHANGE THEIR PRECEDENCE

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE += OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND INCREMENT BOTH REAL AND IMAGINARY PARTS OF THE FIRST BY THOSE OF THE SECOND

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE += OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND INCREMENT BOTH REAL AND IMAGINARY PARTS OF THE FIRST BY THOSE OF THE SECOND

Real Part	Imaginary Part
3	5

OBJECT A (LHS)

+=

Real Part	Imaginary Part
7	6

OBJECT B (RHS)

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE += OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND INCREMENT BOTH REAL AND IMAGINARY PARTS OF THE FIRST BY THOSE OF THE SECOND

Real Part	Imaginary Part
3	5

OBJECT A (LHS)

+=

Real Part	Imaginary Part
7	6

OBJECT B (RHS)

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE += OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND INCREMENT BOTH REAL AND IMAGINARY PARTS OF THE FIRST BY THOSE OF THE SECOND

Real Part	Imaginary Part
3+7	5+6

OBJECT A (LHS)

OBJECT B (RHS) IS UNCHANGED

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE += OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND INCREMENT BOTH REAL AND IMAGINARY PARTS OF THE FIRST BY THOSE OF THE SECOND

Real Part	Imaginary Part
10	11

OBJECT A (LHS)

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
class ComplexNumber
{
private:
   float realPart;
   float complexPart;
public:
   ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
   {
      cout << "Inside the 2-argument constructor" << endl;
   }
   ComplexNumber& operator+=(const ComplexNumber &rhs)
   {
      this->realPart += rhs.realPart;
      this->complexPart += rhs.complexPart;
      return *this;
   }
```

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

USING THE OPERATOR

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

IS A BIT TRICKY, SO LET'S TAKE IT LINE-BY-LINE

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

AN OPERATOR IS JUST LIKE A FUNCTION - SO AN OPERATOR SIGNATURE IS JUST LIKE A FUNCTION SIGNATURE!

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

FOR THE OPERATOR +=, THE FUNCTION NAME EXPECTED BY THE C++ COMPILER IS `operator+=`

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

THE RHS SHOULD NEVER BE CHANGED, SO ITS SHOULD BE A CONST REFERENCE (PREFERABLE TO A PASS-BY-VALUE, AS WE DISCUSSED)

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c(9,11);
(a += b) += c;
```

BECAUSE OPERATIONS LIKE THIS MUST WORK TOO!

THIS WORKS FOR INTS, SO SHOULD WORK FOR USER-DEFINED CLASSES TOO

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c(9,11);
(a += b) += c;
```

BECAUSE OPERATIONS LIKE THIS MUST WORK TOO!

THIS WORKS FOR INTS, SO SHOULD WORK FOR USER-DEFINED CLASSES TOO

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c(9,11);
```

```
(a.operator+=(b)).operator+=(c);
```

BECAUSE OPERATIONS LIKE

IF THIS HAD RETURNED CONST

THIS MUST WORK TOO!

THIS WORKS FOR INTS, SO SHOULD WORK FOR USER-DEFINED CLASSES TOO

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c(9,11);
```

IF THIS HAD RETURNED CONST

```
(a.operator+=(b)).operator+=(c);
```

THE CHAINED OPERATION WOULD FAIL

BECAUSE OPERATIONS LIKE THIS MUST WORK TOO:

THIS WORKS FOR INTS, SO SHOULD WORK FOR USER-DEFINED CLASSES TOO

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

BUT!! THE RETURN VALUE OF += SHOULD BE JUST A REFERENCE, NOT A CONST REFERENCE!

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

IN OTHER WORDS, THE OPERATOR+= MUST BE ABLE TO SUPPORT OPERATOR CHAINING

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

ALSO THE LHS (OBJECT ON WHICH THIS METHOD IS BEING CALLED) IS MODIFIED, SO WE MUST RETURN A REFERENCE TO SELF

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

THE ACTUAL += BITS ARE SIMPLE ENOUGH...

EXAMPLE 44: LEARN HOW TO CORRECTLY OVERLOAD THE += OPERATOR

DEFINING THE OPERATOR

```
ComplexNumber& operator+=(const ComplexNumber &rhs)
{
  this->realPart += rhs.realPart;
  this->complexPart += rhs.complexPart;
  return *this;
}
```

USING THE OPERATOR

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

USING THE OPERATOR

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c(9,11);
(a += b) += c;
```

BECOMES SO SIMPLE AND INTUITIVE ONCE IT HAS BEEN DEFINED RIGHT!

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE + OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND RETURN A NEW OBJECT WITH THE SUM OF THE 2 OBJECTS

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE + OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND RETURN A NEW OBJECT WITH THE SUM OF THE 2 OBJECTS

Real Part	Imaginary Part
3	5

OBJECT A (RHS)

+

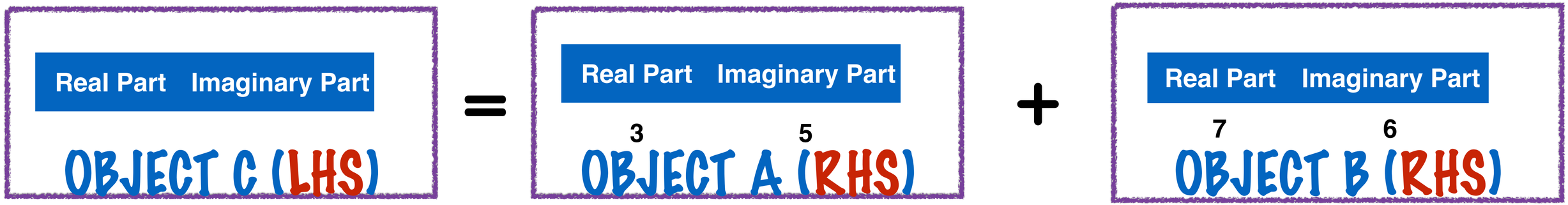
Real Part	Imaginary Part
7	6

OBJECT B (RHS)

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE + OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND RETURN A NEW OBJECT WITH THE SUM OF THE 2 OBJECTS



EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE + OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND RETURN A NEW OBJECT WITH THE SUM OF THE 2 OBJECTS

Real Part	Imaginary Part
3+7	5+6

OBJECT C (LHS)

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

LET'S SAY WE HAVE A COMPLEX NUMBER CLASS, WITH REAL AND IMAGINARY PARTS

THE + OPERATOR WILL TAKE 2 COMPLEX NUMBERS, AND RETURN A NEW OBJECT WITH THE SUM OF THE 2 OBJECTS

Real Part	Imaginary Part
10	11

OBJECT C (LHS)

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
class ComplexNumber
{
private:
   float realPart;
   float complexPart;
public:
   ComplexNumber(double c, double r) : realPart(r) , complexPart(c)
   {
     cout << "Inside the 2-argument constructor" << endl;
   }
   const  ComplexNumber operator+(const ComplexNumber &rhs) const
   {
     ComplexNumber result(this->realPart + rhs.realPart,
                          this->complexPart + rhs.complexPart);
     return result;
   }
```

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

USING THE OPERATOR

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c = a + b;
```

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

IS A BIT TRICKY, SO LET'S TAKE IT LINE-BY-LINE

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

AN OPERATOR IS JUST LIKE A FUNCTION - SO AN OPERATOR SIGNATURE IS JUST LIKE A FUNCTION SIGNATURE!

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

FOR THE OPERATOR +, THE FUNCTION NAME EXPECTED BY THE C++ COMPILER IS `operator+`

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

THE RHS SHOULD NEVER BE CHANGED, SO ITS SHOULD BE A CONST REFERENCE (PREFERABLE TO A PASS-BY-VALUE, AS WE DISCUSSED)

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

THE OBJECT ON WHICH THE OPERATOR+ IS BEING CALLED SHOULD ALSO NOT BE CHANGED, SO MARK IT CONST

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const   ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND **NOT A NON-CONST!**

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
(a + b) = d;
```

BECAUSE OPERATIONS LIKE THIS MUST NOT WORK!

THIS DOES NOT WORKS FOR INTS, SO SHOULD NOT WORK FOR USER-DEFINED CLASSES EITHER

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
```

```
(a + b) = d;
```

BECAUSE OPERATIONS LIKE THIS MUST NOT WORK!

THIS DOES NOT WORKS FOR INTS, SO SHOULD NOT WORK FOR USER-DEFINED CLASSES EITHER

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
```

```
(a.operator+(b)) = d;
```

BECAUSE OPERATIONS LIKE THIS MUST NOT WORK!

THIS DOES NOT WORKS FOR INTS, SO SHOULD NOT WORK FOR USER-DEFINED CLASSES EITHER

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
```

```
(a.operator+(b)) = d;
```

BECAUSE HERE WE ARE TRYING TO MODIFY THE RETURN VALUE!

IF THIS HAD RETURNED A NON-CONST

THIS DOES NOT WORKS FOR INTS, SO SHOULD NOT WORK FOR USER-DEFINED CLASSES EITHER

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
```

IF THIS HAD RETURNED A NON-CONST

```
(a.operator+(b)) = d;
```

THIS LINE WOULD HAVE COMPILED OK (WHEN IT SHOULD THROW AN ERROR)

BECAUSE OPERATIONS LIKE THIS MUST NOT WORK!

THIS DOES NOT WORKS FOR INTS, SO SHOULD NOT WORK FOR USER DEFINED CLASSES EITHER

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

THERE IS JUST NO CORRECT WAY TO RETURN A REFERENCE FROM THIS FUNCTION

THERE IS JUST NO CORRECT WAY TO RETURN A REFERENCE FROM THIS FUNCTION

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

IF THE RESULT IS A STACK VARIABLE, RETURNING A REFERENCE TO IT WILL LEAD TO A GARBAGE VALUE IN THE CALLING CODE

THERE IS JUST NO CORRECT WAY TO RETURN A REFERENCE FROM THIS FUNCTION

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

IF THE RESULT IS A HEAP VARIABLE, RETURNING A REFERENCE TO IT WILL LEAD TO A MEMORY LEAK (WHO WILL FREE THE MEMORY OF A TEMPORARY VARIABLE?)

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

DEFINING THE OPERATOR

```
const   ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

BUT!! THE RETURN VALUE OF + SHOULD BE JUST A CONST, NOT A REFERENCE, AND NOT A NON-CONST!

EXAMPLE 45: LEARN HOW TO CORRECTLY OVERLOAD THE + OPERATOR

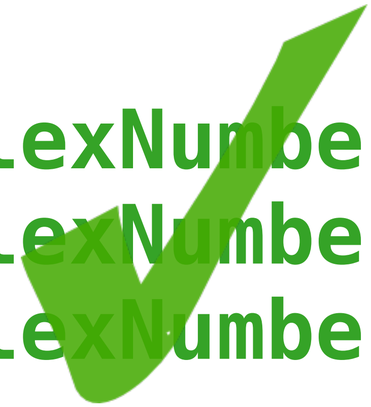
DEFINING THE OPERATOR

```
const  ComplexNumber operator+(const ComplexNumber &rhs) const
{
  ComplexNumber result(this->realPart + rhs.realPart,
                       this->complexPart + rhs.complexPart);
  return result;
}
```

USING THE OPERATOR

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c = a + b;
```

USING THE OPERATOR



```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber c = a + b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber d;
```

X

```
(a + b) = d;
```

BECOMES SO SIMPLE AND INTUITIVE
ONCE IT HAS BEEN DEFINED RIGHT!

EXAMPLE 46: UNDERSTAND THE FUNCTION SIGNATURES OF THE ++ AND -- (PRE-INCREMENT AND POST-INCREMENT OPERATORS)

EXAMPLE 46: UNDERSTAND THE FUNCTION SIGNATURES OF THE ++ AND -- (PRE-INCREMENT AND POST-INCREMENT OPERATORS)

```
ComplexNumber a(3,5);
a++;   // post-increment
++a;   // pre-increment
```

IS EQUIVALENT TO...

```
ComplexNumber a(3,5);
a++;  // post-increment
++a;  // pre-increment
```

IS EQUIVALENT TO...

```
ComplexNumber a(3,5);
a.operator()++;  // post-increment
a.operator()++;  // pre-increment
```

ERRM..HOW CAN 2 FUNCTIONS WITH IDENTICAL SIGNATURES EXIST?

ERRM..HOW CAN 2 FUNCTIONS WITH IDENTICAL SIGNATURES EXIST?

```
ComplexNumber a(3,5);
a.operator(int dummy)++;  // post-increment
a.operator()++;  // pre-increment
```

THEY CAN'T! AND THAT'S WHY THE POST-INCREMENT WILL TAKE IN A DUMMY ARGUMENT

THEY CAN'T! AND THAT'S WHY THE POST-INCREMENT WILL TAKE IN A DUMMY ARGUMENT

```
ComplexNumber a(3,5);
a.operator(int dummy)++;  // post-increment
a.operator()++;  // pre-increment
```

THIS DUMMY ARGUMENT IS NOT PASSED IN, THE C++ COMPILER WILL JUST USE IT IN THE SIGNATURE TO DISAMBIGUATE THE OPERATORS TO CALL

THIS DUMMY ARGUMENT IS NOT PASSED IN, THE C++ COMPILER WILL JUST USE IT IN THE SIGNATURE TO DISAMBIGUATE THE OPERATORS TO CALL

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
{
  cout << "Inside the pre-increment operator" << endl;
  this->realPart++;
  this->complexPart++;
  return *this;
}

// a++ i.e. the post-increment
const ComplexNumber operator++(int)
{
  cout << "Inside the post-increment operator" << endl;
  ComplexNumber result(this->realPart,this->complexPart);
  this->realPart++;
  this->complexPart++;
  return result;
}
```

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
```

```
ComplexNumber a(3,5);
ComplexNumber b = ++a;   // pre-increment
```

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
```

```
ComplexNumber a(3,5);
ComplexNumber b = a++;   // pre-increment
```

PRE-INCREMENT = "INCREMENT-THEN-FETCH"

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
```

```
ComplexNumber a(3,5);
ComplexNumber b = ++a;   // pre-increment
```

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
```

```
ComplexNumber a(3,5);
ComplexNumber b = a++;   // pre-increment
```

POST-INCREMENT = "FETCH-THEN-INCREMENT"

PRE-INCREMENT = "INCREMENT-THEN-FETCH"

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
ComplexNumber a(3,5);
ComplexNumber b = ++a;   // pre-increment
      a = (4,6); b = (4,6);
```

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
ComplexNumber a(3,5);
ComplexNumber b = a++;   // pre-increment
      a = (4,6); b = (3,5);
```

POST-INCREMENT = "FETCH-THEN-INCREMENT"

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
```

WHY DOES PRE-INCREMENT RETURN A REFERENCE?

```
++a = b;
```

SO THAT STATEMENTS LIKE THIS CAN COMPILE (THEY DO FOR INTS)

PRE-INCREMENT = "INCREMENT-THEN-FETCH"

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
ComplexNumber a(3,5);
ComplexNumber b = ++a;   // pre-increment
      a = (4,6); b = (4,6);
```

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
ComplexNumber a(3,5);
ComplexNumber b = a++;   // pre-increment
      a = (4,6); b = (3,5);
```

POST-INCREMENT = "FETCH-THEN-INCREMENT"

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
```

WHY DOES POST-INCREMENT RETURN A CONST?

X `a++++;`

SO THAT STATEMENTS LIKE THIS DO NOT COMPILE (THEY DON'T FOR INTS)

ALSO, THIS CAN NOT RETURN A REFERENCE FOR THE SAME REASON THAT `operator+()` CAN NOT

ONCE WE KNOW THE RETURN TYPES, THE ACTUAL IMPLEMENTATION IS NOT THAT HARD..

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
{
  cout << "Inside the pre-increment operator" << endl;
  this->realPart++;
  this->complexPart++;
  return *this;
}

// a++ i.e. the post-increment
const ComplexNumber operator++(int)
{
  cout << "Inside the post-increment operator" << endl;
  ComplexNumber result(this->realPart,this->complexPart);
  this->realPart++;
  this->complexPart++;
  return result;
}
```

ONCE WE KNOW THE RETURN TYPES, THE ACTUAL IMPLEMENTATION IS NOT THAT HARD..

```
// ++a i.e. the pre-increment
ComplexNumber& operator++()
{
  cout << "Inside the pre-increment operator" << endl;
  this->realPart++;
  this->complexPart++;
  return *this;
}
```

PRE-INCREMENT = "INCREMENT-THEN-FETCH"

ONCE WE KNOW THE RETURN TYPES, THE ACTUAL IMPLEMENTATION IS NOT THAT HARD..

POST-INCREMENT = "FETCH-THEN-INCREMENT"

```
// a++ i.e. the post-increment
const ComplexNumber operator++(int)
{
  cout << "Inside the post-increment operator" << endl;
  ComplexNumber result(this->realPart,this->complexPart);
  this->realPart++;
  this->complexPart++;
  return result;
}
```

EXAMPLE 47: UNDERSTAND THE HOW AND WHY OF THE ASSIGNMENT OPERATOR (`operator=`)

EXAMPLE 47: UNDERSTAND THE HOW AND WHY OF THE ASSIGNMENT OPERATOR (operator=)

THE ASSIGNMENT OPERATOR (operator=) IS VERY IMPORTANT, AND RATHER COMPLICATED.

SO - LET'S BREAK IT DOWN

THE ASSIGNMENT OPERATOR (`operator=`) IS VERY IMPORTANT

WHY?

BECAUSE YOU HAVE TO WRITE IT MORE OFTEN THAN MOST OTHER OPERATORS (OR FUNCTIONS)

WHY?

BECAUSE THE C++ COMPILER PROVIDES A DEFAULT IMPLEMENTATION WHICH IS OFTEN WRONG

BECAUSE THE C++ COMPILER PROVIDES A DEFAULT IMPLEMENTATION WHICH IS OFTEN WRONG

WHAT? WHY? HOW?

FOR EVERY CLASS, THE C++ COMPILER PROVIDES AN ASSIGNMENT OPERATOR AS A CONVENIENCE

```
ComplexNumber a(3,4);
ComplexNumber b(1,2);
a = b;
```

FOR EVERY CLASS, THE C++ COMPILER PROVIDES AN ASSIGNMENT OPERATOR AS A CONVENIENCE

```
ComplexNumber a(3,4);
ComplexNumber b(1,2);
a = b;
```

THIS DEFAULT IMPLEMENTATION WILL LITERALLY COPY OVER THE CONTENTS OF THE MEMORY LOCATIONS OCCUPIED BY `b` INTO THOSE OF `a`

IF THE OBJECTS CONTAIN ANY DYNAMICALLY ALLOCATED MEMORY, THIS IS -

WRONG! WRONG! WRONG

FOR EVERY CLASS, THE C++ COMPILER PROVIDES AN ASSIGNMENT OPERATOR AS A CONVENIENCE

IF THE OBJECTS CONTAIN ANY DYNAMICALLY ALLOCATED MEMORY, THIS IS - WRONG! WRONG! WRONG

REASON #1: ALL OF THE DYNAMICALLY ALLOCATED MEMORY INSIDE A WILL LEAK

REASON #2: ALL OF THE POINTERS WILL ONLY BE "SHALLOW COPIED", WHICH IS PROBABLY WRONG

FOR EVERY CLASS, THE C++ COMPILER PROVIDES AN ASSIGNMENT OPERATOR AS A CONVENIENCE

IF THE OBJECTS CONTAIN ANY DYNAMICALLY ALLOCATED MEMORY, THIS IS - WRONG! WRONG! WRONG

REASON #1: ALL OF THE DYNAMICALLY ALLOCATED MEMORY INSIDE A WILL LEAK

REASON #2: ALL OF THE POINTERS WILL ONLY BE "SHALLOW COPIED", WHICH IS PROBABLY WRONG

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

OKEY-DOKEY! NOW THAT WE KNOW WHY ITS IMPORTANT (AND A BIT COMPLICATED)..

LET'S FIGURE OUT HOW TO WRITE THIS :-)

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
       delete[] studentName;
       if (rhs.studentName != NULL)
           {
           // 3.  Allocate some memory to hold the contents of rhs
           studentName = new char[50];
           // 4.  Copy the values from rhs into this instance
           strcpy(studentName,rhs.studentName);
           }
      }
    // 5.  Return *this
    return *this;
  }
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
       delete[] studentName;
       if (rhs.studentName != NULL)
           {
           // 3.  Allocate some memory to hold the contents of rhs
           studentName = new char[50];
           // 4.  Copy the values from rhs into this instance
           strcpy(studentName,rhs.studentName);
           }
      }
    // 5.  Return *this
    return *this;
  }
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
          {
          // 3.  Allocate some memory to hold the contents of rhs
          studentName = new char[50];
          // 4.  Copy the values from rhs into this instance
          strcpy(studentName,rhs.studentName);
          }
      }
    // 5.  Return *this
    return *this;
  }
```

OUR CLASS HAS A POINTER VARIABLE, SO WE WILL HAVE TO IMPLEMENT THE ASSIGNMENT OPERATOR

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
```

AND! THERE IT IS IN ALL ITS GLORY

```
Student & operator=(const Student &rhs)
{
  // 1. check for self-assignment
  if (this != &rhs)
    {
      // 2.  Deallocate any memory that Student is using internally
      cout << "Freeing up memory for string " << studentName << endl;
     delete[] studentName;
     if (rhs.studentName != NULL)
         {
         // 3.  Allocate some memory to hold the contents of rhs
         studentName = new char[50];
         // 4.  Copy the values from rhs into this instance
         strcpy(studentName,rhs.studentName);
         }
    }
  // 5.  Return *this
  return *this;
}
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

IT RETURNS A REFERENCE (NOT-CONST), SO THAT ASSIGNMENT CAN BE CHAINED..

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
          {
          // 3.  Allocate some memory to hold the contents of rhs
          studentName = new char[50];
          // 4.  Copy the values from rhs into this instance
          strcpy(studentName,rhs.studentName);
          }
      }
    // 5.  Return *this
    return *this;
  }
```

```
a = b = c = d;
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        delete[] studentName;
        if (rhs.studentName != NULL)
          {
            // 3.  Allocate some memory to hold the contents of rhs
            studentName = new char[50];
            // 4.  Copy the values from rhs into this instance
            strcpy(studentName, rhs.studentName);
          }
      }
    // 5. Return *this
    return *this;
  }
```

IT RETURNS A REFERENCE (NOT-CONST), SO THAT ASSIGNMENT CAN BE CHAINED..

```
a = b = c = d;
a.operator=(b.operator=(c.operator=(d)));
```

(ASSIGNMENT IS RIGHT ASSOCIATIVE)

SO, A REFERENCE MUST BE RETURNED SO THAT B TAKES THE CORRECT VALUE. (NOT NEEDED FOR C OR A!)

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
          {
          // 3.  Allocate some memory to hold the contents of rhs
          studentName = new char[50];
          // 4.  Copy the values from rhs into this instance
          strcpy(studentName,rhs.studentName);
          }
      }
    // 5.  Return *this
    return *this;
  }
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

ELSE STEP 2 WILL CAUSE DISASTER

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
        // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
       delete[] studentName;
       if (rhs.studentName != NULL)
           {
           // 3.  Allocate some memory to hold the contents of rhs
           studentName = new char[50];
           // 4.  Copy the values from rhs into this instance
           strcpy(studentName,rhs.studentName);
           }
      }
    // 5.  Return *this
    return *this;
  }
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
    {
      // 2.  Deallocate any memory that Student is using internally
      cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
        {
        // 3.  Allocate some memory to hold the contents of rhs
        studentName = new char[50];
        // 4.  Copy the values from rhs into this instance
        strcpy(studentName,rhs.studentName);
        }
    }
    // 5.  Return *this
    return *this;
  }
```

NOW, FOLLOW 5 SIMPLE STEPS!

BTW, YOU MUST CHECK FOR SELF-ASSIGNMENT LIKE THIS

NOT LIKE THIS

```
if (*this != rhs)
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

CLEAN UP YOUR EXISTING POINTER (COPY-PASTE FROM DESTRUCTOR :-))

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
       // 2.  Deallocate any memory that Student is using internally
        cout << "Freeing up memory for string " << studentName << endl;
       delete[] studentName;
       if (rhs.studentName != NULL)
           {
           // 3.  Allocate some memory to hold the contents of rhs
           studentName = new char[50];
           // 4.  Copy the values from rhs into this instance
           strcpy(studentName,rhs.studentName);
           }
      }
    // 5.  Return *this
    return *this;
  }
```

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
       // 2.  Deallocate any memory that Student is using internally
       cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
        {
        // 3.  Allocate some memory to hold the contents of rhs
        studentName = new char[50];
        // 4.  Copy the values from rhs into this instance
        strcpy(studentName,rhs.studentName);
        }
      }
    // 5.  Return *this
    return *this;
  }
```

SET UP YOUR NEW STATE (COPY PASTE FROM COPY CONSTRUCTOR! :-))

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
       // 2.  Deallocate any memory that Student is using internally
       cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
         {
         // 3.  Allocate some memory to hold the contents of rhs
         studentName = new char[50];
         // 4.  Copy the values from rhs into this instance
         strcpy(studentName,rhs.studentName);
         }
      }
    // 5.  Return *this
    return *this;
  }
```

SET UP YOUR NEW STATE (COPY PASTE FROM COPY CONSTRUCTOR! :-))

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

NOW, FOLLOW 5 SIMPLE STEPS!

```
class Student
{
private:
   char * studentName;
public:
  Student & operator=(const Student &rhs)
  {
    // 1. check for self-assignment
    if (this != &rhs)
      {
       // 2.  Deallocate any memory that Student is using internally
       cout << "Freeing up memory for string " << studentName << endl;
      delete[] studentName;
      if (rhs.studentName != NULL)
          {
          // 3.  Allocate some memory to hold the contents of rhs
          studentName = new char[50];
          // 4.  Copy the values from rhs into this instance
          strcpy(studentName,rhs.studentName);
          }
      }
    // 5.  Return *this
    return *this;
  }
```

RETURN * THIS FOR USUAL ASSIGNMENT CHAINING SEMANTICS

SO - ANYTIME YOUR OBJECTS HAVE ANY DYNAMICALLY ALLOCATED MEMORY

YOU NEED TO WRITE YOUR OWN VERSION OF THE ASSIGNMENT OPERATOR

```
class Student
{
private:
   char * studentName;
public:
```

AND! THERE IT IS IN ALL ITS GLORY

```
Student & operator=(const Student &rhs)
{
  // 1. check for self-assignment
  if (this != &rhs)
    {
      // 2.  Deallocate any memory that Student is using internally
      cout << "Freeing up memory for string " << studentName << endl;
     delete[] studentName;
     if (rhs.studentName != NULL)
         {
         // 3.  Allocate some memory to hold the contents of rhs
         studentName = new char[50];
         // 4.  Copy the values from rhs into this instance
         strcpy(studentName,rhs.studentName);
         }
    }
  // 5.  Return *this
  return *this;
}
```

EXAMPLE 48: OVERLOADING THE << AND >> OPERATORS

BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE
NEXT FEW MINUTES AHEAD OF TIME
INTO A BUFFER

RECAP FROM C

BUFFER

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE NEXT
FEW MINUTES AHEAD OF TIME INTO A BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

A BUFFER IS USED WHENEVER THERE IS A
DIFFERENCE IN THE SPEED BETWEEN THE
INPUT AND OUTPUT DEVICES

RECAP FROM C

BUFFER

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE NEXT
FEW MINUTES AHEAD OF TIME INTO A BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

IN C ALL I/O IS DONE
WITH STREAMS

A BUFFER IS USED WHENEVER THERE IS A
DIFFERENCE IN THE SPEED BETWEEN THE
INPUT AND OUTPUT DEVICES

RECAP FROM C

BUFFER

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE NEXT
FEW MINUTES AHEAD OF TIME INTO A BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

IN C ALL I/O IS DONE
WITH STREAMS

A STREAM IS LIKE A CONVEYOR BELT THAT CARRIES BYTES OF DATA TO BE PROCESSED

A BUFFER IS USED WHENEVER THERE IS A
DIFFERENCE IN THE SPEED BETWEEN THE
INPUT AND OUTPUT DEVICES

RECAP FROM C

BUFFER

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE NEXT
FEW MINUTES AHEAD OF TIME INTO A BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

IN C ALL I/O IS DONE
WITH STREAMS

A STREAM IS LIKE A
CONVEYOR BELT THAT
CARRIES BYTES OF DATA
TO BE PROCESSED

THERE ARE 3 PREDEFINED STREAMS AVAILABLE IN STDIO.H

A BUFFER IS USED WHENEVER THERE IS A
DIFFERENCE IN THE SPEED BETWEEN THE
INPUT AND OUTPUT DEVICES

RECAP FROM C

BUFFER

IF YOU ARE WATCHING A VIDEO ONLINE,
(MAYBE THIS ONE.. :))
THE WEB SERVICE WILL DOWNLOAD THE NEXT
FEW MINUTES AHEAD OF TIME INTO A BUFFER

A BUFFER IS A TEMPORARY STORAGE AREA

IN C ALL I/O IS DONE
WITH STREAMS

A STREAM IS LIKE A
CONVEYOR BELT THAT
CARRIES BYTES OF DATA
TO BE PROCESSED

THERE ARE 3 PREDEFINED
STREAMS AVAILABLE IN
STDIO.H

this is where inputs typed from your keyboard go

stdin — the standard input stream

stdout — the standard output stream

this is where outputs printed to screen go

stderr — the standard error stream

this is where error messages and diagnostics are stored - until you

A BUFFER IS USED WHENEVER THERE IS A
DIFFERENCE IN THE SPEED BETWEEN THE
INPUT AND OUTPUT DEVICES

RECAP FROM C

A BUFFER IS A TEMPORARY STORAGE AREA

IN C ALL I/O IS DONE WITH STREAMS

RECAP FROM C

A BUFFER IS A TEMPORARY STORAGE AREA

IN C++ ALL I/O IS DONE WITH STREAMS

SCREEN IO

`cout` `clog` `cerr` `cin`

STANDARD OUTPUT STREAM - WRITE TO THIS STREAM TO WRITE TO SCREEN

use the `<<` operator

STANDARD INPUT STREAM - READ FROM HERE TO READ FROM KEYBOARD

use the `>>` operator

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{ // C Translation: printf("Hi there! This is a C++ world.\n");
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    // C Translation: Standard output stream
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

// C Translation: send-to

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;   // C Translation:"\n"
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;  // C Translation: scanf("%d", &someNumber);
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;  // C Translation: standard input stream
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;  // C Translation: get-from
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

FLASHBACK WITHIN C++

EXAMPLE 48: OVERLOADING THE << AND >> OPERATORS

EXAMPLE 48: OVERLOADING THE << AND >> OPERATORS

THE << AND >> OPERATORS MUST BE EXTERNAL OPERATORS (RESIDE OUTSIDE THE CLASS)

WHY?

```
ComplexNumber c(3,4);
int x = 5, y = 7;
cout << x << c << y << endl;
```

THE << AND >> OPERATORS MUST BE EXTERNAL OPERATORS (RESIDE OUTSIDE THE CLASS)

WHY?

SO THAT CODE LIKE THIS CAN WORK

```
ComplexNumber c(3,4);
int x = 5, y = 7;
cout << x << c << y << endl;
```

THIS HIGHLIGHTED BIT, IN PARTICULAR, REQUIRES THAT THE FIRST OPERAND NOT BE AN OBJECT..ONLY POSSIBLE WITH AN EXTERNAL OPERATOR.

INTERNAL OR EXTERNAL OPERATORS?

ANOTHER ISSUE TO CONSIDER WITH OPERATORS IS WHETHER THEY SHOULD BE MEMBER FUNCTIONS, OR GLOBAL (USUALLY FRIENDS)*

*NOT ALL EXTERNAL OPERATORS NECESSARILY HAVE TO BE FRIENDS, BUT IT IS OFTEN THE CASE

SOME OPERATORS HAVE TO BE EXTERNAL

SOME OPERATORS HAVE TO BE EXTERNAL

WHY? BECAUSE AN INTERNAL OPERATOR WILL ONLY WORK IF ONE OF THE OPERANDS IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber::operator+=(*(a->this),b);
```

FLASHBACK WITHIN C++

SOME OPERATORS HAVE TO BE EXTERNAL

WHY? BECAUSE AN INTERNAL OPERATOR WILL ONLY WORK IF ONE OF THE OPERANDS IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a += b;
```

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

WILL ONLY WORK IF **a** IS AN OBJECT

```
ComplexNumber a(3,5);
ComplexNumber b(7,6);
ComplexNumber::operator+=(*(a->this),b);
```

FLASHBACK WITHIN C++

SO THIS CODE WILL NOT WORK WITH AN INTERNAL OPERATOR

```
double a(3,5);
ComplexNumber b(7,6);
a += b;
```

BECAUSE **a** IS A BUILT-IN TYPE, NOT AN OBJECT

```
double a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
double a(3,5);
ComplexNumber b(7,6);
double::operator+=(*(a->this),b);
```

SO THIS CODE WILL NOT WORK WITH AN INTERNAL OPERATOR

```
double a(3,5);
ComplexNumber b(7,6);
a += b;
```

BECAUSE **a** IS A BUILT-IN TYPE, NOT AN OBJECT

```
double a(3,5);
ComplexNumber b(7,6);
a.operator+=(b);
```

```
double a(3,5);
ComplexNumber b(7,6);
double::operator+=(*(a->this),b);
```

THIS IS WHY SOME OPERATORS, SUCH AS << AND >> HAVE TO BE EXTERNAL

THE << AND >> OPERATORS MUST BE EXTERNAL OPERATORS (RESIDE OUTSIDE THE CLASS)

WHY?

SO THAT CODE LIKE THIS CAN WORK

```
ComplexNumber c(3,4);
int x = 5, y = 7;
cout << x << c << y << endl;
```

THIS HIGHLIGHTED BIT, IN PARTICULAR, REQUIRES THAT THE FIRST OPERAND NOT BE AN OBJECT..ONLY POSSIBLE WITH AN EXTERNAL OPERATOR.

SO - OVERLOADING THESE OPERATORS HAS 2 STEPS

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
  ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " << complexNumber.complexPart << endl;
  return outputStream;
}

  istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

STEP 2: IF NEEDED, DECLARE THE FUNCTIONS TO BE FRIENDS (INSIDE YOUR CLASS)

```
friend ostream& operator<< (ostream& s, const ComplexNumber& c);
friend istream& operator>> (istream& s, ComplexNumber& c);
```

SO - OVERLOADING THESE OPERATORS HAS 2 STEPS

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
  ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " << complexNumber.complexPart << endl;
  return outputStream;
}

  istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

STEP 2: IF NEEDED, DECLARE THE FUNCTIONS TO BE FRIENDS (INSIDE YOUR CLASS)

```
friend ostream& operator<< (ostream& s, const ComplexNumber& c);
friend istream& operator>> (istream& s, ComplexNumber& c);
```

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

DON'T BE INTIMIDATED BY THESE SIGNATURES!

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

ostream AND **istream** ARE SIMPLY THE CLASSES REPRESENTING THE IO STREAMS WE JUST DISCUSSED

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

THE OPERATORS TAKE IN THE STREAM..

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

..THE OBJECT TO BE WRITTEN TO (OR READ FROM) THE STREAM..

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

THE >> OPERATOR MODIFIES THE OBJECT, SO IT IS NOT CONST, BUT THE << OPERATOR IS CONST

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

THE OPERATORS READ FROM (OR WRITE TO) THE OBJECT - ACCESSING PRIVATE MEMBERS SINCE THEY ARE FRIENDS

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " <<
complexNumber.complexPart << endl;
  return outputStream;
}

istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

AND FINALLY RETURN THE STREAM THAT THEY JUST READ FROM/WROTE TO

SO - OVERLOADING THESE OPERATORS HAS 2 STEPS

STEP 1: WRITE THE OPERATORS OUTSIDE THE CLASS (GLOBALLY, IN THE STD NAMESPACE)

```
  ostream& operator<< (ostream& outputStream, const ComplexNumber& complexNumber)
{
  outputStream << "real = " << complexNumber.realPart << " complex = " << complexNumber.complexPart << endl;
  return outputStream;
}

  istream& operator>> (istream& inputStream, ComplexNumber& complexNumber)
{
  inputStream >> complexNumber.realPart >> complexNumber.complexPart;
  return inputStream;
}
```

STEP 2: IF NEEDED, DECLARE THE FUNCTIONS TO BE FRIENDS (INSIDE YOUR CLASS)

```
friend ostream& operator<< (ostream& s, const ComplexNumber& c);
friend istream& operator>> (istream& s, ComplexNumber& c);
```

STEP 2: IF NEEDED, DECLARE THE FUNCTIONS TO BE FRIENDS (INSIDE YOUR CLASS)

```
friend ostream& operator<< (ostream& s, const ComplexNumber& c);
friend istream& operator>> (istream& s, ComplexNumber& c);
```

IF THE OPERATORS CAN AVOID ACCESSING PRIVATE DATA, THEY SHOULD NOT BE MARKED AS FRIENDS

AVOID DECLARING FRIENDS UNLESS YOU ABSOLUTELY HAVE TO

USING THE OPERATORS IS PRETTY STRAIGHTFORWARD NOW

```
ComplexNumber a(3,5);
cout << "Initial value of a" << a << endl;
cin >> a;
cout << "Final value of a" << a << endl;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example48.cpp
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Inside the 2-argument constructor
Initial value of areal = 5 complex = 3

4
9
Final value of areal = 4 complex = 9
```