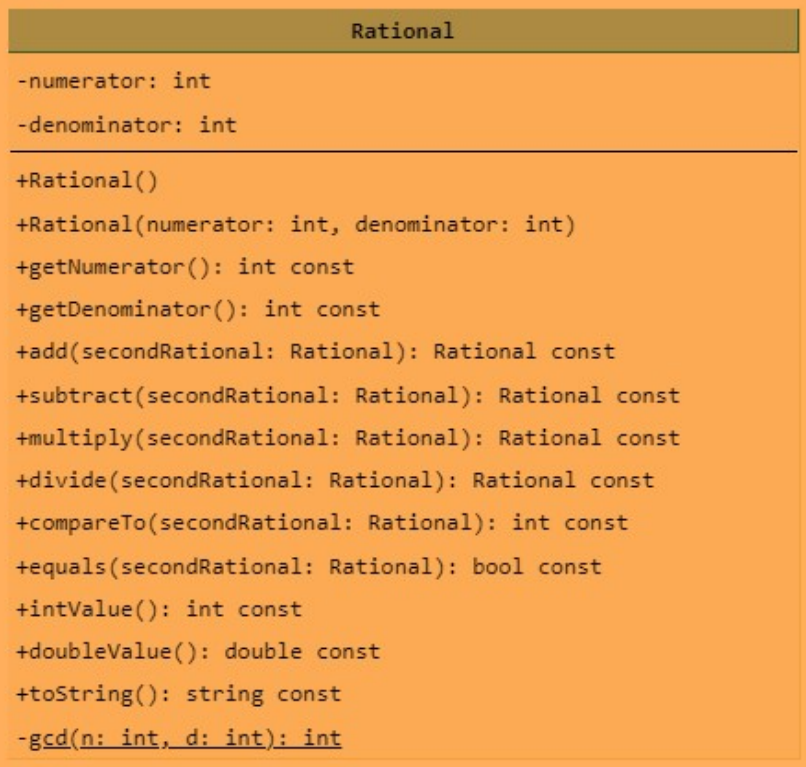
14.2 The Rational Class

Monday, April 10, 2023 9:12 AM

- Section defines the Rational class for modeling rational numbers
- Rational number has a numerator and denominator, form a/b, where a = numerator, b= denominator
- Rational numb cant have denom = 0, but can have numer = 0
- Every int I is equivalent to rational numb i/1
- Rational numbs used in exact computations involving fractions
  - Like 1/3 = 0.333333...
  - Its not precise in floating pt format using data type double / float
  - To get exact result, use rational numbs
- C++ gives data types for integers and floating pt, but not rational numbs, so we design own class to rep rational numbs
- A Rational number can be repped using 2 data fields:
  - Numerator
  - Denominator
- Can make rational number w/ specific numer and denom, or can set default as numer=0 and denom=1
- Can +, -, /, *, and compare rational numbs
- Can also convert a rational number into an integer, floating-pt val, or string
- **Figure 14.1**



The properties, constructors, and functions of the `Rational` class are illustrated in UML.

- A rational number has a numerator and denominator, many equivalent rational numbers
- To reduce rational number to its lowest terms (2/6 == 3/9 == 1/3, 1/3 is the lowest term), need to find GCD (greatest common divisor) of the absolute values of its numerator and denominator and then divide both numerator and denominator by this val

- Can use the fn for finding GCD of 2 ints

```
#ifndef RATIONAL_H
#define RATIONAL_H
#include <string>
using namespace std;

class Rational
{
public:
  Rational();
  Rational(int numerator, int denominator);
  int getNumerator() const;
  int getDenominator() const;
  Rational add(const Rational& secondRational) const;
  Rational subtract(const Rational& secondRational) const;
  Rational multiply(const Rational& secondRational) const;
  Rational divide(const Rational& secondRational) const;
  int compareTo(const Rational& secondRational) const;
  bool equals(const Rational& secondRational) const;
  int intValue() const;
  double doubleValue() const;
  string toString() const;

private:
  int numerator;
  int denominator;
  static int gcd(int n, int d);
};

#endif
```

LiveExample 14.2 TestRationalClass.cpp

Source Code Editor:

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

int main()
{
  // Create and initialize two rational numbers r1 and r2.
  Rational r1(4, 2);
  Rational r2(2, 3);

  // Test toString, add, subtract, multiply, and divide
  cout << r1.toString() << " + " << r2.toString() << " = "
    << r1.add(r2).toString() << endl;
  cout << r1.toString() << " - " << r2.toString() << " = "
    << r1.subtract(r2).toString() << endl;
  cout << r1.toString() << " * " << r2.toString() << " = "
    << r1.multiply(r2).toString() << endl;
  cout << r1.toString() << " / " << r2.toString() << " = "
    << r1.divide(r2).toString() << endl;

  // Test intValue and double
  cout << "r2.intValue()" << " is " << r2.intValue() << endl;
  cout << "r2.doubleValue()" << " is " << r2.doubleValue() << endl;

  // Test compareTo and equal
  cout << "r1.compareTo(r2) is " << r1.compareTo(r2) << endl;
  cout << "r2.compareTo(r1) is " << r2.compareTo(r1) << endl;
  cout << "r1.compareTo(r1) is " << r1.compareTo(r1) << endl;
  cout << "r1.equals(r1) is "
       << (r1.equals(r1) ? "true" : "false") << endl;
  cout << "r1.equals(r2) is "
       << (r1.equals(r2) ? "true" : "false") << endl;

  return 0;
}
```

```
command>cl TestRationalClass.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestRationalClass
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
r2.intValue() is 0
r2.doubleValue() is 0.666667
r1.compareTo(r2) is 1
r2.compareTo(r1) is -1
r1.compareTo(r1) is 0
r1.equals(r1) is true
r1.equals(r2) is false

command>
```

- The main fn makes 2 rational numbers (r1, r2), and displays the results r1 + r2, -, *, and /
  - To do the stuff, uses r1.add(r2), .subtract, .multiply, and .divide (respectively)
- The intValue() fn displays the int value of r2, the doubleValue displays the double value of r2
- Invoking r1.compareTo(r2) returns 1 bc r1 > r2
- Invoking r2.compareTo(r1) returns -1 bc r1 > r2
- Invoking r1.compareTo(r1) returns 0 bc r1 = r1
- Invoking r1.equals(r1) returns true bc r1 == r1
- Invoking r1.equals(r2) returns false bc r1 != r2

```
#include "Rational.h"
#include <cstdlib> // For the abs function
Rational::Rational(){
  numerator = 0;
  denominator = 1;
}
Rational::Rational(int numerator, int denominator){
  int factor = gcd(numerator, denominator);
  this->numerator = ((denominator > 0) ? 1 : -1) * numerator / factor;
  this->denominator = abs(denominator) / factor;
}
int Rational::getNumerator() const{
  return numerator;
}
int Rational::getDenominator() const{
  return denominator;
}
// Find GCD of two numbers
int Rational::gcd(int n, int d) {
  int n1 = abs(n);
  int n2 = abs(d);
  int gcd = 1;
  for (int k = 1; k <= n1 && k <= n2; k++){
    if (n1 % k == 0 && n2 % k == 0)
      gcd = k;
  }
  return gcd;
}
Rational Rational::add(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator() + denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}
Rational Rational::subtract(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator() - denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}
Rational Rational::multiply(const Rational& secondRational) const{
  int n = numerator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}
```

```
Rational Rational::divide(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator();
  int d = denominator * secondRational.numerator;
  return Rational(n, d);
}
int Rational::compareTo(const Rational& secondRational) const{
  Rational temp = subtract(secondRational);
  if (temp.getNumerator() < 0)
    return -1;
  else if (temp.getNumerator() == 0)
    return 0;
  else
    return 1;
}
bool Rational::equals(const Rational& secondRational) const{
  if (compareTo(secondRational) == 0)
    return true;
  else
    return false;
}
int Rational::intValue() const{
  return getNumerator() / getDenominator();
}
double Rational::doubleValue() const{
  return 1.0 * getNumerator() / getDenominator();
}
string Rational::toString() const{
  if (denominator == 1)
    return to_string(numerator); // See Ch7 for to_string
  else
    return  to_string(numerator) + "/" + to_string(denominator);
}
```

- The rational number is encapsulated in a Rational object, internally a rational number is repped in lowest terms, and numerator determines its sign, denominator is always positive
- The gcd() fn is private, not used by clients
- The gcd() is only for internal use by the Rational class, the gcd() fn is also static since not dependent on any particular Rational object
- 2 Rational objects can interact w/ each other to do +, -, *, and / operations
- These methods return a new Rational object
  - The math formula for these ops:
  - a/b + c/d = (ad + bc)/(bd) (e.g., 2/3 + 3/4 = (2*4 + 3*3)/(3*4) = 17/12)

    a/b - c/d = (ad - bc)/(bd) (e.g., 2/3 - 3/4 = (2*4 - 3*3)/(3*4) = -1/12)

    a/b * c/d = (ac)/(bd) (e.g., 2/3 * 3/4 = (2*3)/(3*4) = 6/12 = 1/2)

    a/b / c/d = (ad)/(bc) (e.g., (2/3) / (3/4) = (2*4)/(3*3) = 8/9)
  - The abs(x) fn defined in standard C++ library, return absolute val of x
- 2 Rational objects can do these fn, returns a new Rational object
- The compareTo(&secondRational) fn compares this rational numb to the other rational umb
  - First subtracts the 2nd rational from this rational, saves result in temp
  - Returns –1, 0, or 1 if temp's numerator is less than, equal to, or greater than 0
- The equals(&secondRational) fn utilizes the compareTo fn to compare this rational number to the other one, if the fn returns 0, the equals fn returns true, else false
- Fns intvalue and doubleVal return and int / double (respectively)
- The toString() fn returns a string rep of a Rational object in form numerator/denominator, or just numerator if denominator = 1
- The to_string is new in C++11
- The numerator and denominator are repped using 2 vars, can rep them also using an array of 2 ints
- The signature of the public fns in the Rational class are not changed, tho internal reps of a rational numb is changed, good illustration of idea that data fields of a class should be kept private so as to encapsulate the implementation of the class from the use of the class

- Which of the following statements creates a default Rational object?

  - Rational r1(1, 2);
  - Rational r1();
  - ✓ Rational r1;

- What will be displayed by the following code?

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

int main()
{
  cout << Rational().toString();

  return 0;
}
```

  - 0/0
  - 0/1
  - ✓ 0
  - The code has a compile error.

What will be displayed by the following code?

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

int main()
{
  Rational r1(1, 2);
  Rational r2(2, 4);
  cout << r1.equals(r2);

  return 0;
}
```

- true
- false
- 1
- 0

14.3 Operator Functions

Monday, April 10, 2023 9:35 PM

- Most of the operations in C++ can be defined as fns to perform desirable ops
- Convenient to compare 2 string objects using intuitive syntax
  - Like string1 < string2
- Can compare 2 Rational objects using similar syntax
  - Like r1 < r2
- Can define a special fn called the operator fn in the class
- The operator fn is like reg fn, but must be names w/ keyword operator followed by actual operator
  - ```
    bool operator<(const Rational& secondRational) const
    ```
  - This fn header defines the < operator fn that returns true if the Rational object is less than the secondRational
  - Invoke this fn by using
    ```
    r1.operator<(r2)
    ```
  - or simply
    ```
    r1 < r2
    ```
- To use this operator, have to add the fn header for operator< in the public section in the .h file and implement the fn in the .cpp file
- ```
  bool Rational::operator<(const Rational& secondRational) const
  {
    // compareTo is already defined Rational.h
    if (compareTo(secondRational) < 0)
      return true;
    else
      return false;
  }
  ```
-

The following code

```
Rational r1(4, 2);
Rational r2(2, 3);
cout << "r1 < r2 is " << (r1.operator<(r2) ? "true" : "false");
cout << "\nr1 < r2 is " << ((r1 < r2) ? "true" : "false");
cout << "\nr2 < r1 is " << (r2.operator<(r1) ? "true" : "false");
```

displays

```
r1 < r2 is false
r1 < r2 is false
r2 < r1 is true
```

- Note that r1.operator<(r2) is same as r1 < r2 (2nd is simplier, so preferred)
- C++ allow some overload operators

Table 14.1

Operators That Can Be Overloaded

+	-	*	/	%	^	&	|	~	!	=
<	>	+=	-=	*=	/=	%=	^=	&=	|=	<<
>>	>>=	<<=	==	!=	<=	>=	&&	||	++	--
->*	,	->	[]	()	new	delete				

Table 14.2

Operators That Cannot Be Overloaded

?:	.	.*	::

- C++ defines the operator precedence and associativity, cant change the operator precedence and associativity by overloading
- Most operators are binary operators, some uniary, cant change numb of operands by overloading
  - / is binary
  - ++ is unary
- Other ex: overload the + binary op in the Rational class, this is the fn header in Rational.h
- `Rational operator+(const Rational& secondRational) const`
- It implements the fn in Rational.cpp like

```
Rational Rational::operator+(const Rational& secondRational) const
{
  // add is already defined Rational.h
  return add(secondRational);
}
```

-
```
Rational Rational::operator+(const Rational& secondRational) const
{
  // add is already defined Rational.h
  return add(secondRational);
}
```
-
The following code

```
Rational r1(4, 2);
Rational r2(2, 3);
cout << "r1 + r2 is " << (r1 + r2).toString() << endl;
```

displays

```
r1 + r2 is 8/3
```
-
-
The signature for the < operator function for comparing two Rational objects is _____.

- [x] bool operator<(const Rational& secondRational)
- [ ] bool <operator(const Rational& secondRational)
- [ ] bool operator<(const Rational* secondRational)
- [ ] bool operator(<)(const Rational& secondRational)

-
Which of the following operators cannot be overloaded?

- [ ] +
- [ ] +=
- [ ] >
- [ ] &&
- [x] ?:

-

- Which of the following is wrong to add Rational objects r1 to r2?

  - r2 = r2.operator+=(r1);
  - r2 = r1.operator+=(r2);
  - r2 = r2.add(r1);
  - ✓ r1 += r2;
  - r2 += r1;

  Good job!

  r1 += r2; is wrong, because it adds r2 to r1.

14.4 Overloading the Subscript Operator []

Tuesday, April 11, 2023 10:29 AM

- The subscript operator [] commonly defined to access and mod a data field/an element in an object
- C++, pair of [] called subscript operator
- Used to access array elements and elements in a string object and a vector object
- Can overload this operator to access the contents of the object if desirable
  - Might want to access the numerator and denominator of Rational object r using r[0] and r[1]
- First give incorrect solution to overload the [] operator:
  - In header file Rational.h, fn header looks like:
  - ```
    int operator[](int index);
    ```
  - Implement fn in Rational.cpp:
  - ```
    1  int Rational::operator[](int index)
    2  {
    3    if (index == 0)
    4      return numerator;
    5    else
    6      return denominator;
    7  }
    8
    ```
  - Ln 1 is kinda correct
  - The following code

    ```
    Rational r(2, 3);
    cout << "r[0] is " << r[0] << endl;
    cout << "r[1] is " << r[1] << endl;
    ```

    displays

    ```
    r[0] is 2
    r[1] is 3
    ```
  - Set numerator and denominator like array assignment?
  - ```
    r[0] = 5;
    r[1] = 6;
    ```
  - No, error that looks like this
  - ```
    lvalue required in r[0] and r[1]
    ```
- In C++, lvalue refers to a named item persists beyond a single expression, rvalue refers to an unnamed val that disappears after it used in the expression

- All vars and named objects are lvalues, literal values and temporary objects are rvalues
  - Like x and y are lvalues while 4 and 5 are rvalues
  - ```
    int x = 4 * 5; // x is lvalue, 4 and 5 are rvalue
    int y = x + 6 * 7; // x and y are lvalue and 6 and 7 are rvalue
    ```
  - Like s is an lvalue, and string("abc") is an rvalue, string("abc") is an unnamed object string object
  - ```
    string s("Welcome"); // s is lvalue
    s = string("abc"); // string("abc") is an rvalue
    ```
- So how make r[0] and r[1] and lvalue so can assign a value to r[0] and r[1]?, just define the [] operator to return a ref of the var, aka return-by-reference
- Add fn header to Rational.h
- ```
  int& operator[](int index);
  ```
- Implement in Rational.cpp
- ```
  int& Rational::operator[](int index)
  {
    if (index == 0)
      return numerator;
    else
      return denominator;
  }
  ```
- Ln 1 is correct
- Pass-by-reference and return-by-reference is same concept
  - Pass - … , formal param and actual param are aliases
  - Return - … , fn returns an alias to a var, so fn can be used on lft side of an assignment statement
- So, in this fn, if index = 0, fn returns alias of var numerator, if index = 1, fn returns alias of var denominator, fn can now be used as an lvalue
  - Fn does not check bounds of the index, learn later, more robust if check and thro exception if index isn't 0/1

- The following code

```
Rational r(2, 3);
r[0] = 5; // Set numerator to 5
r[1] = 6; // Set denominator to 6
cout << "r[0] is " << r[0] << endl;
cout << "r[1] is " << r[1] << endl;
cout << "r.doubleValue() is " << r.doubleValue() << endl;
```

displays

```
r[0] is 5
r[1] is 6
r.doubleValue() is 0.833333
```

- In r[0], r is an object and 0 is the arg to the memb fn [], when r[0] used as an expression, returns a vlaue for the numerator, when r[0] used on left side of the assignment operator, its an alias for var numerator, so r[0] = 5 assigns 5 to numerator
- The [] operator fns as both accessor and mutator, like r[0] is accessor and r[0] = value is mutator
- For convenience, call a fn operator that returns a reference an lvalue operator, many other operators like +=, -=, *=, /=, and %= are also lvalue operators
- What is the correct signature for overloading the subscript operator []?

  - int Rational::operator[](const int& index)
  - ✓ int& operator[](const int& index);
  - &int operator[](const int& index);
  - int operator&[](const int& index);

14.5 Overloading Augmented Assignment operators

Tuesday, April 11, 2023 10:48 AM

- Can define the augmented assignment operators as fns to return a value by reference
- C++ has augmented assignment operators:
  - +=, for adding
  - -=, for subtracting
  - *=, u know them
  - /=,
  - and %=
- Can overload these operators in the Rational class
- Augmented operators can be used as lvalues
  - like
  -
```
int x = 0;
(x += 2) += 3;
```
  - Is legal, so augmented assignment operators are lvalue operators and u should overload them by return by reference
- Ex: overloading the addition assignment op, +=
- The fn header in the header file Rational.h
-
```
Rational& operator+=(const Rational& secondRational);
```
- And the implementation in the Rational.cpp file
-
```
1  Rational& Rational::operator+=(const Rational& secondRational)
2  {
3    *this = add(secondRational);
4    return *this;
5  }
6
```
- Ln 3 invokes the add fn to add the calling Rational object w/ the 2nd Rational object, result copied to the calling object *this in ln3, calling object is returned in ln 4
- For example, the following code

```
Rational r1(2, 4);
Rational r2 = r1 += Rational(2, 3);
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

displays

```
r1 is 7/6
r2 is 7/6
```

-
- The += operator is implemented incorrectly in the following code. Which of the following statements is false?

```
Rational Rational::operator+=(const Rational& secondRational)
{
  this->add(secondRational);
  return this;
}
```

-
  - There are multiple errors in the code. One error is on "return this". Replace it by return (*this).
  - There are multiple errors in the code. One error is on "this->add(secondRational)". Replace it by *this = this->add(secondRational).
  - There are multiple errors in the code. One error is on the return type. Replace Rational by Rational&.
  - ✓ There are multiple errors in the code. One error is on "Rational::", remove it.
-

14.6 Overloading the Unary Operations

Tuesday, April 11, 2023 3:19 PM

- Unary + and – ops can be overloaded
- The + and – are unary operators, can be overloaded, since unary op ops on 1 operand that is the calling object itself, the unary fn operator has no parameters
- Ex: overloading the – operator
  - Add fn in the header file, Rational.h
  - ```
    Rational operator-()
    ```
  - Implement in the Rational.cpp
  - ```
    Rational Rational::operator-()
    {
      return Rational(-numerator, denominator);
    }
    ```
- Negating a Rational object is same as negating its numerator
- Note that the negating operator returns a new Rational, the calling object itself isnt changed
- The following code

```
1  Rational r2(2, 3);
2  Rational r3 = -r2;  // Negate r2
3  cout << "r2 is " << r2.toString() << endl;
4  cout << "r3 is " << r3.toString() << endl;
5
```

displays

```
r2 is 2/3
r3 is -2/3
```

- What is the correct signature for the overloaded unary operator +?
  - Rational Rational : : operator+(const Rational& r)
  - ✓ Rational Rational : : operator+()
  - Rational Rational : : operator<+>(const Rational& r)
  - Rational Rational : : operator(+)(const Rational& r)
-
-

14.7 Overloading the ++ and – Operators

Tuesday, April 11, 2023 3:31 PM

- The preincrement, predecrement, postincrement, and postdecrement operators can be overloaded
- The ++ and – operators can be prefix/postfix
  - Prefix ++var/--var first adds/subtracts 1 from var then uses the new val in var in the expression
  - Posfix var++/var-- adds/subtracts 1 from the var, uses old value in var in expression tho
- ++ and -- used correcly in code below

```
Rational r2(2, 3);
Rational r3 = ++r2; // Prefix increment
cout << "r3 is " << r3.toString() << endl;
cout << "r2 is " << r2.toString() << endl;

Rational r1(2, 3);
Rational r4 = r1++; // Postfix increment
cout << "r1 is " << r1.toString() << endl;
cout << "r4 is " << r4.toString() << endl;
```

- should display

```
r3 is 5/3
r2 is 5/3
r1 is 5/3
r4 is 2/3
```

- In last line of code, r4 stores og value of r1
- C++ distinguishes prefix ++/-- ops from postfix by using dummy param of the int type and defines the prefix ++ fn operator w/ no param like:

```
Rational& operator++(); // Preincrement
```

-

```
Rational operator++(int dummy); // Postincrement
```

- Note that the prefix ++ and – ops are lvalue ops, but the postfix ++ and – ops are not, so this code is legal:

```
int x = 1;
++x = 4;
```

- But the following code is not allowed:

```
int x = 1;
x++ = 4;
```

- Prefix and postfix ++ op fns for the Rational numbs can be implemented as follows

```
1  // Prefix increment
2  Rational& Rational::operator++()
3  {
4    numerator += denominator;
5    return *this;
6  }
7
8  // Postfix increment
9  Rational Rational::operator++(int dummy)
10 {
11   Rational temp(numerator, denominator);
12   numerator += denominator;
13   return temp;
14 }
15
```

- In prefix ++ fn, ln 4 adds the denom to the numerator, this is the new numerator for the calling object after adding 1 to the Rational object, ln 5 returns the calling object
- In the postfix ++ fn, ln 11 makes a temporary Rational object to store the og calling object, ln 12 increments the calling object, ln 13 returns the og calling object
- What is the correct signature for the overloaded postfix ++ operator?
  - Rational operator++(Rational& r)
  - Rational operator++()
  - ✓ Rational operator++(int dummy)
  - Rational operator++(int& dummy)

14.8 friend Functions and friend Classes

Friday, April 14, 2023 9:48 AM

- Can define a friend function/ friend class to enable it to access private members in another clas
- C++ lets overload stream ops (<< is insertion, >> is extraction)
- These ops usually need to be implemented as friend nonmemb fns, bc may access private data fields in a class
- The friend fns and friend classes to prep to overload these ops
- Private membs of a class cant be accessed from outside the class, but sometimes convenient to allow some trusted fns and classes to access a class's private membs
- C++ lets use friend keyword to define friend fns and friend classes so that these trusted fns and classes can access another class's private membs
- ```
  #ifndef DATE_H
  #define DATE_H
  class Date
  {
  public:
    Date(int year, int month, int day)
    {
      this->year = year;
      this->month = month;
      this->day = day;
    }

    friend class AccessDate; // AccessDate is a friend of Date

  private:
    int year;
    int month;
    int day;
  };

  #endif
  ```
- The AccessDate class is defined as a friend class for the Date class, so can directly access private data field year, month, and day from the AccessDate class
  ```

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

class AccessDate
{
public:
  static void p()
  {
    Date birthDate(2010, 3, 4);
    birthDate.year = 2000; // Access private data in Date
    cout << birthDate.year << endl;
  }
};

int main()
{
  AccessDate::p(); // Invoke p() in AccessDate

  return 0;
}
```

Automatic Check | Compile/Run | Reset | Answer

Execution Result:

```
command>cl TestFriendClass.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestFriendClass
2000

command>
```

- The AccessDate class is defined, a Date object is made in the class, since AccessDate is a friend class of Date class, the private data in a Date object can be accessed in the AccessDate class
- The main fn invokes the static fn AccessDate::p()

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

class Date
{
public:
  Date(int year, int month, int day)
  {
    this->year = year;
    this->month = month;
    this->day = day;
  }
  friend void p(); // p() is a friend function of Date

private:
  int year;
  int month;
  int day;
};

void p()
{
  Date date(2010, 5, 9);
  date.year = 2000;
  cout << date.year << endl;
}

int main()
{
  p();

  return 0;
}
```

```
command>cl TestFriendFunction.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestFriendFunction
2000

command>
```

- This is ex. Of use of friend fn, the pgrm defines the Date class w/ a friend fn p, which is not a member of the Date class but can access the private data in Date, in fn p, a Date object is made, and private field data year is modded

Define and implement the following `Window` class:

- `int` data members, `width` and `height`.
- a constructor that accepts two `int` parameters (`width` followed by `height`) and uses them to initialize the data members.
- a friend function, `areSameSize`, that accepts two `Window` objects and returns a `bool` indicating if they are of the same size. Two `Window` objects are of the same size if their widths and heights match.

```
class Window{
    private:
        int width;
        int height;
    public:
        Window(int width, int height){
            this->width = width;
            this->height = height;
        }
        friend bool areSameSize(Window wind1, Window wind2){
            if ((wind1.width == wind2.width)&&(wind1.height == wind2.height)){
                return true;
            }else{
                return false;
            }
        }
};
```

Which of the following statements is false?

- Private members of a class cannot be accessed from outside of the class unless from a friend class or a friend function.
- C++ enables you to use the friend keyword to declare friend functions and friend classes for a class.
- ✓ friend functions and classes of a class must be declared inside the class.
- friend functions and classes of a class must be declared outside the class.

Good job!

Analyze the following code:

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

class Date
{
  friend void p();

private:
  int year;
  int month;
  int day;
};

void p()
{
  Date date;
  date.year = 2000;
  cout << date.year;
}

int main()
{
  p();
  return 0;
}
```

- The program has a compile error because year is a private data field in Date.
- The program has a runtime error since year is private.
- ✓ The program will have a compile error if the line friend void p() is deleted.
- Since year is private, you cannot access it using date.year in function p().

14.9 Overloading the << and >> Operators

Sunday, April 16, 2023 9:40 AM

- The stream extraction (>>) and insertion (<<) operators can be overloaded for performing input and output ops
- So far, for Rational object, need toString() fn to return a string rep for the Rational object and then display the string
  - Like to display a Rational object r,
  - `cout << r.toString();`
  - But would be nice to just do
  - `cout << r;`
- Stream insertion op (<<) and stream extraction op (>>) are like other binary ops in C++
  - The cout << r is actually same as << (cout, r) or operator<<(cout, r)
  - So r1 + r2 op has 2 Rational objects, both instances of the Rational class, so can overload the + op as a member fn w/ r2 as the arg,
  - But statement cout<< r
  - The operator is << w/ 2 operands cout and r
    - First is instance of the ostream class
    - Other is instance of Rational class
  - C++ req u overload the << op as a non-member fn
- Since fn may need to access the privare data in the Rational class, can define the fn as a friend fn of the Rational class in the Rational.h header file
- ```
  // This is placed in the header file
  friend ostream& operator<<(ostream& out, const Rational& rational);
  ```
- This fn returns a reference of ostream bc u may need to use the << op in a chain of expressions
- `cout << r1 << " followed by " << r2;`
- This is same as
- `cout << r1 << " followed by " << r2;`
  - I honestly don't see the difference, so yea
- For this to work, cout<< r1 must return a reference of ostream, so the fn << can be implemented as follows
- ```
  // This is placed in the implementation file
  ostream& operator<<(ostream& out, const Rational& rational)
  {
    out << rational.numerator << "/" << rational.denominator;
    return out;
  }
  ```
- Can declare and implement the friend fn all in the Rational.h header file like
- ```
  // This combined declaration and implementation is placed inside the header file
  friend ostream& operator<<(ostream& out, const Rational& rational) {
        out << rational.numerator << "/" << rational.denominator;
        return out;
  }
  ```
- Same to overload the >> op, define the fn header in the Rational.h header file
- `friend istream& operator>>(istream& in, Rational& rational);`
- Implement in Rational.cpp
  ```

```
// This is placed in the implementation file
istream& operator>>(istream& in, Rational& rational)
{
  cout << "Enter numerator: ";
  in >> rational.numerator;
  cout << "Enter denominator: ";
  in >> rational.denominator;
  return in;
}
```

- Test pgrm

```
1  Rational r1, r2;
2  cout << "Enter first rational number" << endl;
3  cin >> r1;
4
5  cout << "Enter second rational number" << endl;
6  cin >> r2;
7
8  cout << r1 << " + " << r2 << " = " << r1 + r2 << endl;
9
```

```
Enter first rational number
Enter numerator: 1
Enter denominator: 2
Enter second rational number
Enter numerator: 3
Enter denominator: 4
1/2 + 3/4 is 5/4
```

- Ln 3 reads values to a rational object from cin, ln 8 r1 + r2 evaluated to a new rational number, then sent to cout
- The operator<< and operator>> are defined friend bc implementation accesses the private membs of the Rational class
- If mod their implementation to not access the private membs, dont need to define these fn as friend of Rational

- Assume the existence of a `Window` class with `int` data members `width` and `height`.

  Overload the << operator for the `Window` class-- i.e.,

  write a nonmember `ostream`-returning function that accepts a reference to an `ostream` object and a constant reference to a `Window` object and sends the following to the `ostream`:

  `'a (width x height) window'` (without the quotes and with `width` and `height` replaced by the actual `width` and `height` of the window.

  Thus for example, if the window had `width=80` and `height=20`, << would send 'a (80 x 20) window' to the `ostream`.

  Don't forget to have the function return the proper value as well.

  Note: The `Window` class is already in place. Declare and implement a friend << operator function in the `Window` class.

```
friend ostream& operator<<(ostream& out, const Window& window){
    out << "a (" << window.width << " x " << window.height << ") window";
    return out;
}
```

- Assume the existence of a `Window` class with `int` data members `width` and `height`.

  Overload the >> operator for the `Window` class-- i.e.,
  write a nonmember `istream`-returning function that accepts a reference to an `istream` object and a reference to a `Window` object and reads the `next` two values from the `istream` into the `width` and `height` members respectively.

  Don't forget to have the function return the proper value as well.

  Note: The `Window` class is already in place. Declare and implement a friend >> operator function in the `Window` class.

```
1  friend istream& operator>>(istream& in, Window& window){
2      in >> window.width >> window.height;
3      return in;
4  }
```

- If the << operator does not access the private data fields in Rational, do you still have to declare it friend?

  - Yes
  - ✓ No

  Nice work!

  You don't need to declare it friend if the function does not need to access a class's private data. See the last paragraph in this section.

- What is the correct signature for the overloaded >> operator?
  - friend istream& operator>>(istream& stream, const Rational& rational);
  - friend istream& operator>>(istream& stream, Rational& rational);
  - friend istream operator>>(istream& stream, Rational& rational);
  - friend istream operator>>(istream& stream, const Rational& rational);
-

14.10 Automatic Type conversions

Sunday, April 16, 2023 10:12 AM

- Can define fns to perform auto conversion from an object to a primitive type value and vice versa
- C++ can do certain type conversions auto, can define fns to enable conversions from a Rational object to a primitive type value / vice versa
- 14.10.1 Converting to a Primitive Data type
  - Can add an int value w/ a double value
  - 4 + 5.5
  - So C++ does auto type conversion to convert an int value 4 to a double val 4.0
  - Add rational number w/ an int/double value by defining a fn op to convert an object into int/double
  -
```
Rational::operator double()
{
  return doubleValue(); // doubleValue() already in Rational.h
}
```
  - Must add memb fn header in Rational.h header file
  -
```
operator double();
```
  - Special syntax for defining conversion fns to a primitive type in C++, no return type like a constructor, fn name is the type u want object to be converted to
  - So this
  -
```
1   Rational r1(1, 4);
2   double d = r1 + 5.1;
3   cout << "r1 + 5.1 is " << d << endl;
4
```

displays

```
r1 + 5.1 is 5.35
```

  - Statement in ln 2 adds a rational numb r1 w/ a double value 5.1, since conversion fn is defined to convert a rational number to a double, r1 is converted to a double value 0.25 which then added w/ 5.1
- 14.10.2 Converting to an Object Type
  - A Rational object can be auto converted to a numeric value, numeric value can be auto converted to a Rational object thru fn
  - So header file
  -
```
Rational(int numerator);
```
  - And implement

-
```
Rational::Rational(int numerator)
{
  this->numerator = numerator;
  this->denominator = 1;
}
```
- Given that the + op also overloaded, this code
```
Rational r1(2, 3);
Rational r = r1 + 4; // Automatically converting 4 to Rational
cout << r << endl;
```
- displays
```
14 / 3
```
- When C++ sees r1 + 4, first checks to see if the + op has been overloaded to add a Rational w/ an int, since no fn is defined, sys next searches for the + op to add a Rational w/ another Rational
- Since 4 is an integer, C++ uses the constructor that constructs a Rational object from an integer arg, so C++ does an auto conversion to convert an integer to a Rational object
- This auto conversion possible bc suitable constructor is available, now 2 Rational objects are added using the overloaded + op to return a new Rational object (14 / 3)

- A class can define the conversion fn to convert an object to a primitive type value/ define a conversionconstructor to convert a primitive type val to an object, but not both simultaneously in the class, if both are defined, the compiler will report an ambiguity error
-
What is the correct signature for a function that converts a Rational to double?

- double operator()
- double operator double()
- Rational operator double()
- operator double()

14.11 Defining Nonmember Fns for Overloading Ops

Sunday, April 16, 2023 7:34 PM

- If an operator can be overloaded as a nonmember fn, define it as a nonmember fn to enable implicit type conversions
- C++ can perform certain type conversions auto, can define fns to enable conversions , can add a Rational object r1 w/ an integer like
  - This: r1 + 4
  - Can't like this: 4 + r1
- This bc the left operand is the calling object for the + op, and the left operand must be a Rational object, so 4 as the left operand wont work (its an int), and C++ does not do auto conversion in this case
- To not have this problem, do 2 things
  1. Define and implement this constructor
  - ```
    Rational(int numerator);
    ```
  - Which lets enable the integer to be converted to a Rational object
  2. Define the + op as a nonmember fn in the Rational.h header like
  - ```
    Rational operator+(const Rational& r1, const Rational& r2)
    ```
  - And implement the fn in Rational.cpp like
  - ```
    Rational operator+(const Rational& r1, const Rational& r2)
    {
      return r1.add(r2);
    }
    ```
  - Auto type conversion to the user-defined object also works comparison operators (<, <=, ==, !=, >, >=)
  - Note that the ex for the operator< and operator+ are defined as memb fns in diff section (14.3), so from now on we define them as nonmember fns
-

The signature for the nonmember < operator function for comparing two Rational objects is _____.

- [x] bool operator<(const Rational& r1, const Rational& r2)
- [ ] bool <operator(const Rational& r1, const Rational& r2)
- [ ] bool operator<(Rational r1, Rational r2)
- [ ] bool operator(<)(const Rational& r1, const Rational& r2)

-
    ```

14.12 The Rational Class w/ Overloaded Function Operators

Sunday, April 16, 2023 8:00 PM

- This section revises the Rational class w/ overloaded fn operators
- Next sections intro how to overload fn ops
- Good notes
  - Conversion fns from a class type to a primitive type/ from a primitive type to a class type cannot both be defined in the same class, this bc would cause ambiguity error bc compiler cant decide which conversion to perform; usually converting from a primitive type to a class type is more useful, so will define our Rational class to support auto conversion from a primitive type to the Rational type
  - Most ops can be overloaded wither as memb or non-memb fns; the ++, --, +=, -=, *=, /=, %=, =, and [] ops must be overloaded as memb fns and << and >> ops must be overloaded as nonmemb fns
  - The binary +, -, *, %, /, <, <=, ==, !=, >, and >= ops should be overloaded as nonmemb fns to enable auto type conversion w/ symmetric operands
  - If want the returned object to be used as an lvalue (like used on the left-hand side of the assignment statement), need to define the fn to return a reference, the aug assignment operators +=, -=, *=, /=, and %=, the prefix ++ and – ops, the subscript op [], and the assignment ops = are lvalue ops
- Below, give new header file named RationalWithOperators.h for the Rational class with fn ops
-

```
#ifndef RATIONALWITHOPERATORS_H
#define RATIONALWITHOPERATORS_H
#include <string>
#include <iostream>
using namespace std;

class Rational
{
public:
  Rational();
  Rational(int numerator, int denominator);
  int getNumerator() const;
  int getDenominator() const;
  Rational add(const Rational& secondRational) const;
  Rational subtract(const Rational& secondRational) const;
  Rational multiply(const Rational& secondRational) const;
  Rational divide(const Rational& secondRational) const;
  int compareTo(const Rational& secondRational) const;
  bool equals(const Rational& secondRational) const;
  int intValue() const;
  double doubleValue() const;
  string toString() const;

  Rational(int numerator); // Suitable for type conversion

  // Define function operators for augmented operators
  Rational& operator+=(const Rational& secondRational);
  Rational& operator-=(const Rational& secondRational);
  Rational& operator*=(const Rational& secondRational);
  Rational& operator/=(const Rational& secondRational);

  // Define function operator []
  int& operator[](int index);
```

```
  // Define function operators for prefix ++ and --
  Rational& operator++();
  Rational& operator--();

  // Define function operators for postfix ++ and --
  Rational operator++(int dummy);
  Rational operator--(int dummy);

  // Define function operators for unary + and -
  Rational operator+();
  Rational operator-();

  // Define the << and >> operators
  friend ostream& operator<<(ostream&, const Rational&);
  friend istream& operator>>(istream&, Rational&);

private:
  int numerator;
  int denominator;
  static int gcd(int n, int d);
};

// Define nonmember function operators for relational operators
bool operator<(const Rational& r1, const Rational& r2);
bool operator<=(const Rational& r1, const Rational& r2);
bool operator>(const Rational& r1, const Rational& r2);
bool operator>=(const Rational& r1, const Rational& r2);
bool operator==(const Rational& r1, const Rational& r2);
bool operator!=(const Rational& r1, const Rational& r2);

// Define nonmember function operators for arithmetic operators
Rational operator+(const Rational& r1, const Rational& r2);
Rational operator-(const Rational& r1, const Rational& r2);
Rational operator*(const Rational& r1, const Rational& r2);
Rational operator/(const Rational& r1, const Rational& r2);

#endif
```

- Fns for aug assignment ops (+=, -=, *=, /=), subscript op [], prefix ++, and prefix -- are defined to return a reference
- Stream extraction >> and stream insertion << ops are defined
- Nonmemb fns for comparison ops (>, >=, <, <=, ==, !=) and arithmetic ops (+, -, *, /)

```
#include "RationalWithOperators.h"
#include <cstdlib> // For the abs function

Rational::Rational(){
  numerator = 0;
  denominator = 1;
}

Rational::Rational(int numerator, int denominator){
  int factor = gcd(numerator, denominator);
  this->numerator = (denominator > 0 ? 1 : -1) * numerator / factor;
  this->denominator = abs(denominator) / factor;
}

int Rational::getNumerator() const{
  return numerator;
}

int Rational::getDenominator() const{
  return denominator;
}
```

```
// Find GCD of two numbers
int Rational::gcd(int n, int d) {
  int n1 = abs(n);
  int n2 = abs(d);
  int gcd = 1;

  for (int k = 1; k <= n1 && k <= n2; k++){
    if (n1 % k == 0 && n2 % k == 0)
      gcd = k;
  }

  return gcd;
}
```

○

```
Rational Rational::add(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator() +
    denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::subtract(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator()
    - denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}
```

```
Rational Rational::multiply(const Rational& secondRational) const{
  int n = numerator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::divide(const Rational& secondRational) const{
  int n = numerator * secondRational.getDenominator();
  int d = denominator * secondRational.numerator;
  return Rational(n, d);
}
```

○

```
int Rational::compareTo(const Rational& secondRational) const{
  Rational temp = subtract(secondRational);
  if (temp.getNumerator() < 0)
    return -1;
  else if (temp.getNumerator() == 0)
    return 0;
  else
    return 1;
}
```

```
bool Rational::equals(const Rational& secondRational) const{
  if (compareTo(secondRational) == 0)
    return true;
  else
    return false;
}

int Rational::intValue() const{
  return getNumerator() / getDenominator();
}

double Rational::doubleValue() const{
  return 1.0 * getNumerator() / getDenominator();
}

string Rational::toString() const{
  if (denominator == 1)
    return to_string(numerator); // See Ch7 for to_string
  else
    return  to_string(numerator) + "/" + to_string(denominator);
}

Rational::Rational(int numerator) // Suitable for type conversion{
  this->numerator = numerator;
  this->denominator = 1;
}
// Define function operators for augmented operators
Rational& Rational::operator+=(const Rational& secondRational){
  *this = add(secondRational);
  return *this;
}

Rational& Rational::operator-=(const Rational& secondRational){
  *this = subtract(secondRational);
  return *this;
}

Rational& Rational::operator*=(const Rational& secondRational){
  *this = multiply(secondRational);
  return *this;
}

Rational& Rational::operator/=(const Rational& secondRational){
  *this = divide(secondRational);
  return *this;
}
```

```
// Define function operator []
int& Rational::operator[](int index){
  if (index == 0)
    return numerator;
  else
    return denominator;
}

// Define function operators for prefix ++ and --
Rational& Rational::operator++(){
  numerator += denominator;
  return *this;
}

Rational& Rational::operator--(){
  numerator -= denominator;
  return *this;
}

// Define function operators for postfix ++ and --
Rational Rational::operator++(int dummy){
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}

Rational Rational::operator--(int dummy){
  Rational temp(numerator, denominator);
  numerator -= denominator;
  return temp;
}

// Define function operators for unary + and -
Rational Rational::operator+(){
  return *this;
}

Rational Rational::operator-(){
  return Rational(-numerator, denominator);
}

// Define the output and input operator
ostream& operator<<(ostream& out, const Rational& rational){
  if (rational.denominator == 1)
    out << rational.numerator;
  else
    out << rational.numerator << "/" << rational.denominator;
  return out;
}
```

```
istream& operator>>(istream& in, Rational& rational){
  cout << "Enter numerator: ";
  in >> rational.numerator;

  cout << "Enter denominator: ";
  in >> rational.denominator;
  return in;
}

// Define function operators for relational operators
bool operator<(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) < 0);
}

bool operator<=(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) <= 0);
}

bool operator>(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) > 0);
}

bool operator>=(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) >= 0);
}
```

```
bool operator==(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) == 0);
}

bool operator!=(const Rational& r1, const Rational& r2){
  return (r1.compareTo(r2) != 0);
}

// Define non-member function operators for arithmetic operators
Rational operator+(const Rational& r1, const Rational& r2){
  return r1.add(r2);
}

Rational operator-(const Rational& r1, const Rational& r2){
  return r1.subtract(r2);
}

Rational operator*(const Rational& r1, const Rational& r2){
  return r1.multiply(r2);
}

Rational operator/(const Rational& r1, const Rational& r2){
  return r1.divide(r2);
}
```

- Implements the header file, memb fns for aug assignment ops (I am not writing them again do it yourself), change contents of the calling object
- Have to assign the result of the operation to this ,the comparison op are implemented by invoking r1.compareTo(r2) , the arithmetic ops (no) are implemented by invoking the fns add, subtract … u get the gist

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

int main()
{
  // Create and initialize two rational numbers r1 and r2.
  Rational r1(4, 2);
  Rational r2(2, 3);

  // Test relational operators
  cout << r1 << " > " << r2 << " is " << ((r1 > r2) ? "true" : "false") << endl;
  cout << r1 << " < " << r2 << " is " << ((r1 < r2) ? "true" : "false") << endl;
  cout << r1 << " == " << r2 << " is " <<((r1 == r2) ? "true" : "false") << endl;
  cout << r1 << " != " << r2 << " is " <<((r1 != r2) ? "true" : "false") << endl;

  // Test toString, add, subtract, multiply, and divide operators
  cout << r1 << " + " << r2 << " = " << r1 + r2 << endl;
  cout << r1 << " - " << r2 << " = " << r1 - r2 << endl;
  cout << r1 << " * " << r2 << " = " << r1 * r2 << endl;
  cout << r1 << " / " << r2 << " = " << r1 / r2 << endl;

  // Test augmented operators
  Rational r3(1, 2);
  r3 += r1;
  cout << "r3 is " << r3 << endl;

  // Test function operator []
  Rational r4(1, 2);
  r4[0] = 3; r4[1] = 4;
  cout << "r4 is " << r4 << endl;

  // Test function operators for prefix ++ and --
  r3 = r4++;
  cout << "r3 is " << r3 << endl;
  cout << "r4 is " << r4 << endl;

  // Test function operator for conversion
  cout << "1 + " << r4 << " is " << (1 + r4) << endl;

  return 0;
}
```

```
command>cl TestRationalWithOperators.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestRationalWithOperators
2 > 2/3 is true
2 < 2/3 is false
2 == 2/3 is false
2 != 2/3 is true
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
r3 is 5/2
r4 is 3/4
r3 is 3/4
r4 is 7/4
1 + 7/4 is 11/4

command>
```

Suppose r is a Rational object, in order to perform 4 + r to obtain a Rational object, the Rational class header file must contain:

- conversion function int operator()
- ✓ conversion constructor Rational(int numerator) and non-member function Rational operator+ (const Rational& r1, const Rational& r2)
- member function Rational operator+(const Rational& r1)

•

14.13 Overloading the = Operators

Sunday, April 16, 2023 8:46 PM

- Need to overload the = operator to perform a customized copy operation for an object
- By default, the = operator does a memberwise copy from one object to the other, like this copying r2 to r1

```
Rational r1(1, 2);
Rational r2(4, 5);
r1 = r2;
cout << "r1 is " << r1 << endl;
cout << "r2 is " << r2 << endl;
```

So, the output is

```
r1 is 4/5
r2 is 4/5
```

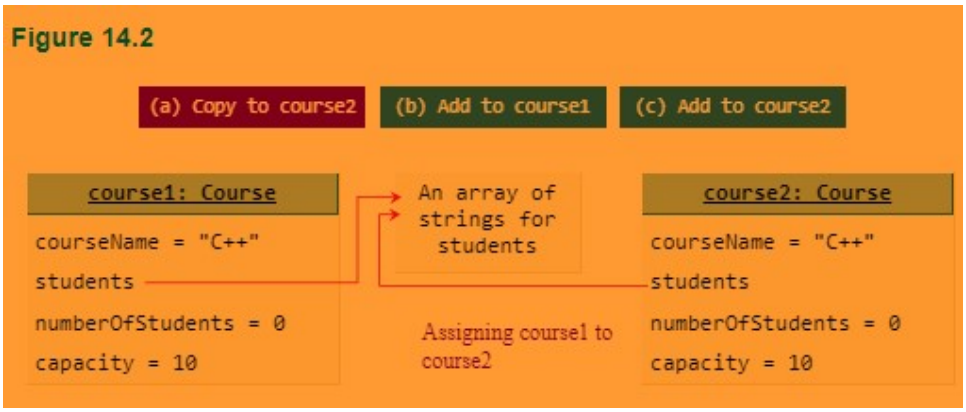
- Behavior of the = op is same as default copy constructor, does shallow copy (data field is pointer to some object, address of the pointer is copied instead of the contents)

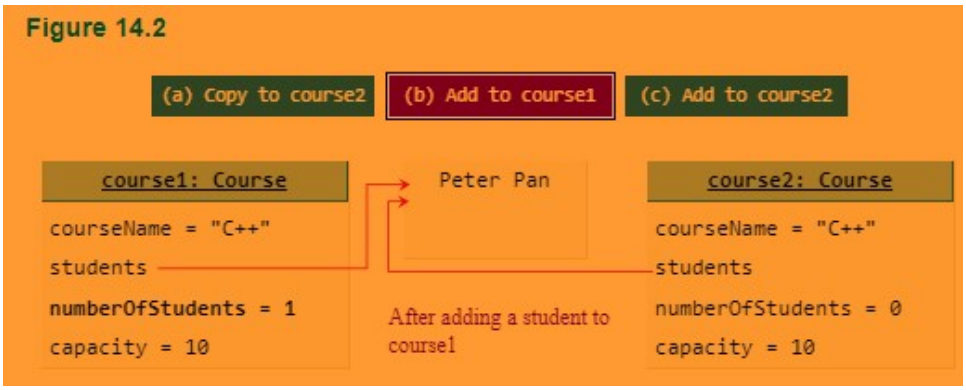
```
#include <iostream>
#include "CourseWithCustomCopyConstructor.h" // See Listing 11.19
using namespace std;

int main()
{
  Course course1("C++", 10);
  Course course2("Java", 14);
  course2 = course1; // Assign course1 to course2

  course1.addStudent("Peter Pan"); // Add a student to course1
  course2.addStudent("Lisa Ma"); // Add a student to course2

  cout << "students in course1: " <<
    course1.getStudents()[0] << endl;
  cout << "students in course2: " <<
    course2.getStudents()[0] << endl;

  return 0;
}
```

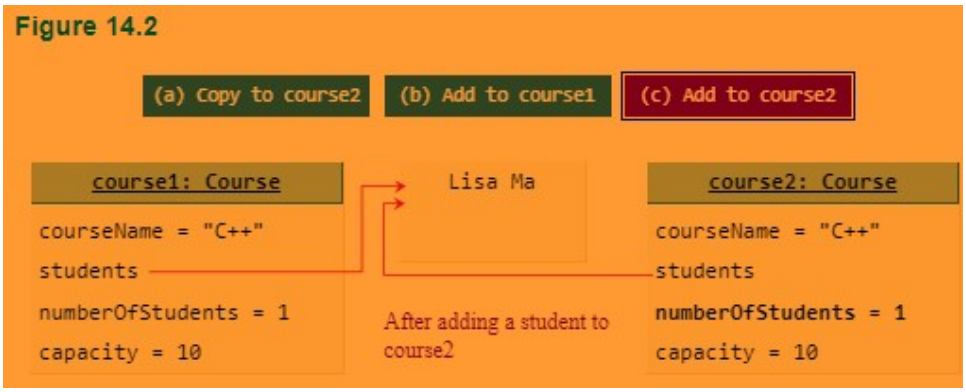
Automatic Check | Compile/Run | Reset | Answer | Choose a Compiler:

Execution Result:

```
command>cl DefaultAssignmentDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>DefaultAssignmentDemo
students in course1: Lisa Ma
students in course2: Lisa Ma

command>
```

- 

- 

- 

- The students array in course1 and course2 are the same after assigning course1 to course2 in ln 9, when add student "peter pan" to course1, "Peter Pan" is assigned to students[numberOfStudents], where numberOfStudents is 0
- After the assignment, numberOfStudents is increased by 1
- When add student "Lisa Ma" to course2, "Lisa Ma" is assigned to students[numberOfStudents], where numberOfStudents is 0
- After the assignment, numberOfStudents is inc by 1
- Now both course1.getStudents()[0] and course2.getStudents()[0] will be "Lisa Ma"

```
1  #ifndef COURSE_H
2  #define COURSE_H
3  #include <string>
4  using namespace std;
5
6  class Course
7  {
8  public:
9    Course(const string& courseName, int capacity);
10   ~Course(); // Destructor
11   Course(Course&); // Copy constructor
12   string getCourseName() const;
13   void addStudent(const string& name);
14   void dropStudent(const string& name);
15   string* getStudents() const;
16   int getNumberOfStudents() const;
17   Course& operator=(const Course& course); // Assignment operator
18
19 private:
20   string courseName;
21   string* students;
22   int numberOfStudents;
23   int capacity;
24 };
25
26 #endif
```

- Ln 9 isn't void bc in C++, allowed to have chained expression assignments, like
- `course1 = course2 = course3;`
- In this statement, course3 is copied to course2, and then returns course2, and then course2 is copied to course1, so the = op must have a valid return value type

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

Course::Course(const string& courseName, int capacity)
{
  numberOfStudents = 0;
  this->courseName = courseName;
  this->capacity = capacity;
  students = new string[capacity];
}

Course::~Course()
{
  delete [] students;
}

string Course::getCourseName() const
{
  return courseName;
}

void Course::addStudent(const string& name)
{
  if (numberOfStudents >= capacity)
  {
    cout << "The maximum size of array exceeded" << endl;
    cout << "Program terminates now" << endl;
    exit(0);
  }

  students[numberOfStudents] = name;
  numberOfStudents++;
}
```

```
void Course::dropStudent(const string& name)
{
  // Left as an exercise
}

string* Course::getStudents() const
{
  return students;
}

int Course::getNumberOfStudents() const
{
  return numberOfStudents;
}

Course::Course(Course& course) // Copy constructor
{
  courseName = course.courseName;
  numberOfStudents = course.numberOfStudents;
  capacity = course.capacity;
  students = new string[capacity];
  for (int i = 0; i < numberOfStudents; i++)
    students[i] = course.students[i];
}
```

```
Course& Course::operator=(const Course& course) // Assignment operator
{
  if (this != &course) // Do nothing with self-assignment
  {
    courseName = course.courseName;
    numberOfStudents = course.numberOfStudents;
    capacity = course.capacity;

    delete [] this->students; // Delete the old array

    // Create a new array with the same capacity as course copied
    students = new string[capacity];
    for (int i = 0; i < numberOfStudents; i++)
      students[i] = course.students[i];
  }

  return *this;
}
```

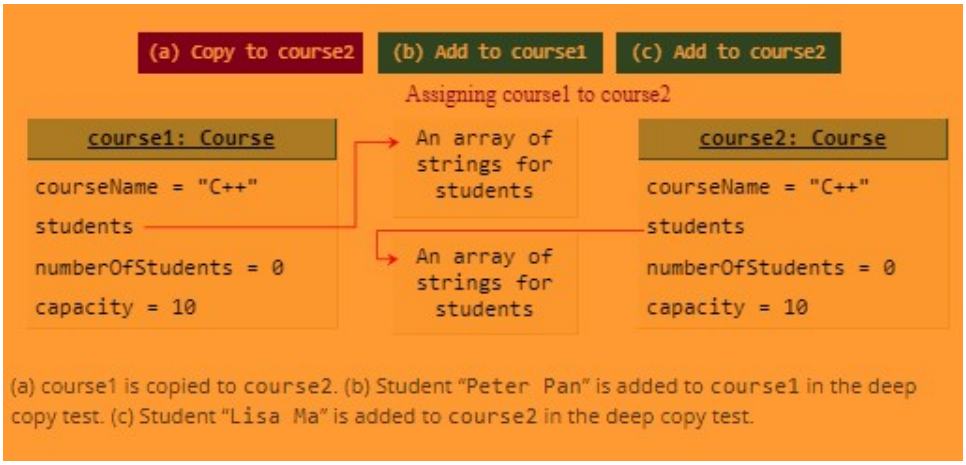
- Look at new test pgrm that uses overloaded = op to copy a Course object, the 2 courses have diff students array

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

void printStudent(const string names[], int size)
{
  for (int i = 0; i < size; i++)
    cout << names[i] << (i < size - 1 ? ", " : " ");
}

int main()
{
  Course course1("C++", 10);
  Course course2("Java", 10);

  // Assign course1 to course2
  course2 = course1;
  course1.addStudent("Peter Pan"); // Add a student to course1
  course2.addStudent("Lisa Ma"); // Add a student to course2

  cout << "students in course1: ";
  printStudent(course1.getStudents(), course1.getNumberOfStudents());
  cout << endl;

  cout << "students in course2: ";
  printStudent(course2.getStudents(), course2.getNumberOfStudents());
  cout << endl;

  return 0;
}
```

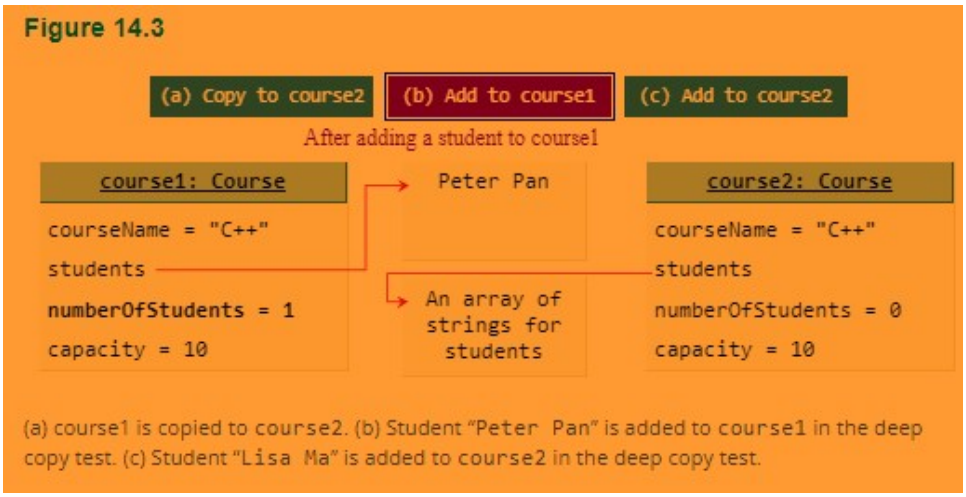
Automatic Check | Compile/Run | Reset | Answer | Choose a Compiler:

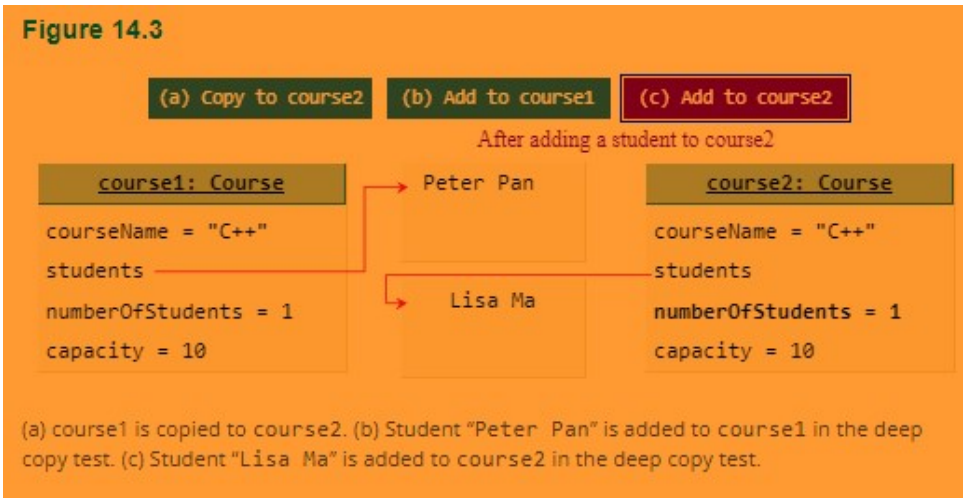
Execution Result:

```
command>cl CustomAssignmentDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>CustomAssignmentDemo
students in course1: Peter Pan
students in course2: Lisa Ma

command>
```



(a) course1 is copied to course2. (b) Student "Peter Pan" is added to course1 in the deep copy test. (c) Student "Lisa Ma" is added to course2 in the deep copy test.

- 

- 

- Note, that like 14.3 a, the students array in course1 and course2 are diff after assigning course1 to course2 in ln 17 as result of performing a deep copy, like 14.3 b
- When add student "Lisa Ma" to couse2, "Lisa Ma" is assigned to students[numberOfStudents], where numberOfStudents is 0, after assignment, numberOfStudents is inc by 1, shown in 14.3 c
- Now students in course1 is "Peter Pan" and in course2 are "Peter Pan" and "Lisa Ma"

- **Note**

  The following statement

  ```
  Course c = c1;
  ```

  is equivalent to

  ```
  Course c(c1);
  ```

  which is different from the following statements

  ```
  Course c; // Create object c using the Course no-arg constructor
  c = c1; // Assign c1 to c
  ```

- **rule of three**
  The copy constructor, the = assignment operator, and the destructor; also known as the Big Three

- **Note**

  The copy constructor, the = assignment operator, and the destructor are called the rule of three, or *the Big Three*. If they are not defined explicitly, all three are created by the compiler automatically. You should customize them if the class contains a pointer data field. If you have to customize one of the three, you should customize the other two as well. In C++11, the Big Three rule is expanded to the Big Five with move constructors and move assignments. See Supplement IV.I, "C++11 Move Constructors and Move Assignment," at **https://liangcpp.pearsoncmg.com/supplement/MoveSemantics.pdf**.

  https://liangcpp.pearsoncmg.com/supplement/MoveSemantics.pdf

- https://liangcpp.pearsoncmg.com/supplement/MoveSemantics.pdf

What is the correct signature for the = operator function in the Course class?

- operator==(const Course& course);
- operator=(const Course& course);
- Course& operator=(const Course& course);
- Course operator=(const Course& course);

14 Summary

Sunday, April 16, 2023 9:28 PM

1. C++ allows you to overload operators to simplify operations for objects.
2. You can overload nearly all operators except ?:, ., .*, and ::
3. You cannot change the operator precedence and associativity by overloading.
4. You can overload the subscript operator [] to access the contents of the object using the if desirable.
5. A C++ function may return a reference, which is an alias for the returned variable.
6. The augmented assignment operators (+=, -=, *=, /=), subscript operator [], prefix ++, and prefix -- operators are lvalue operators. The functions for overloading these operators should return a reference.
7. The friend keyword can be used to give the trusted functions and classes access to a class's private members.
8. The operators +=, -=, *=, /=, %=,[], ++, --, and [] must be overloaded as member functions.
9. The << and >> operators must be overloaded as nonmember functions.
10. The arithmetic operators (+, -, *, /) and comparison operators (>, >=, ==, !=, <, <=) should be implemented as nonmember functions.
11. C++ can perform certain type conversions automatically if appropriate functions and constructors are defined.
12. By default, the memberwise shallow copy is used for the = operator. To perform a deep copy for the = operator, you need to overload the = operator.

*14.7 (*Math: The Complex class*) A complex number has the form $a + bi$, where a and b are real numbers and i is $\sqrt{-1}$. The numbers a and b are known as the real part and imaginary part of the complex number, respectively. You can perform addition, subtraction, multiplication, and division for complex numbers using the following formulas:

$$a + bi + c + di = (a + c) + (b + d)i$$
$$a + bi - (c + di) = (a - c) + (b - d)i$$
$$(a + bi) * (c + di) = (ac - bd) + (bc + ad)i$$
$$(a + bi)/(c + di) = (ac + bd)/(c^2 + d^2) + (bc - ad)i/(c^2 + d^2)$$

You can also obtain the absolute value for a complex number using the following formula:

$$|a + bi| = \sqrt{a^2 + b^2}$$

(A complex number can be interpreted as a point on a plane by identifying the (a, b) values as the coordinates of the point. The absolute value of the complex number corresponds to the distance of the point to the origin, as shown in **Figure 14.4**.)

Figure 14.4

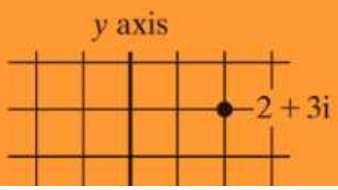
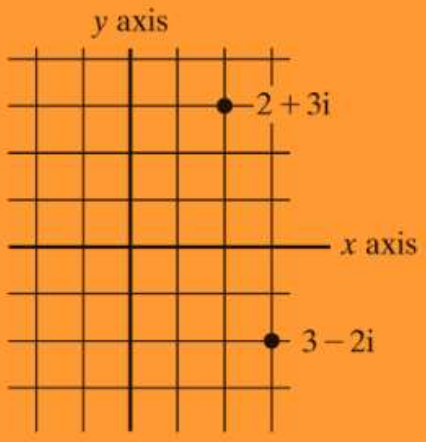


Figure 14.4



A complex number can be interpreted as a point in a plane.

Design a class named `Complex` for representing complex numbers and the functions `add`, `subtract`, `multiply`, `divide`, `abs` for performing complex-number operations, and the `toString` function for returning a string representation for a complex number. The `toString` function returns `a + bi` as a string. If `b` is `0`, it simply returns `a`.

Provide three constructors `Complex(a, b)`, `Complex(a)`, and `Complex()`. `Complex()` creates a `Complex` object for number `0` and `Complex(a)` creates a `Complex` object with `0` for `b`. Also provide the `getRealPart()` and `getImaginaryPart()` functions for returning the real and imaginary part of the complex number, respectively.

Overload the operators `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `[]`, unary `+` and `-`, prefix `++` and `--`, postfix `++` and `--`, `<<`, `>>`.

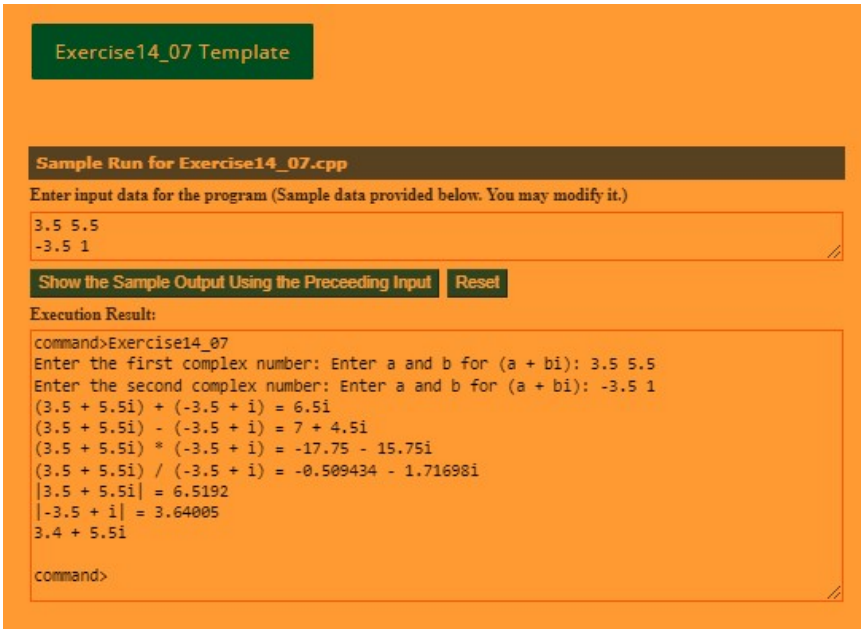
Overload `[]` so that `[0]` returns `a` and `[1]` returns `b`.

Overload the relational operators `<`, `<=`, `==`, `!=`, `>`, `>=` by comparing the absolute values of two complex numbers.

Overload the operators `+`, `-`, `*`, `/`, `<<`, `>>`. `<`, `<=`, `==`, `!=`, `>`, `>=` as nonmember functions.

Write a test program that prompts the user to enter two complex numbers and display the result of their addition, subtraction, multiplication, division, and their absolute values. Use the template in **https://liangcpp.pearsoncmg.com/test/Exercise14_07_5e.txt** to write your code.

Exercise14_07 Template



https://liangcpp.pearsoncmg.com/test/Exercise14_07_5e.txt

```
// Search for "WRITE YOUR CODE" to complete this program
#include <sstream>
#include <cmath>
#include <string>
#include <iostream>
using namespace std;

class Complex
{
public:
  Complex();
  Complex(double a, double b);
  Complex(double a);
  double getRealPart() const;
  double getImaginaryPart() const;
  Complex add(const Complex& secondComplex) const;
  Complex subtract(const Complex& secondComplex) const;
  Complex multiply(const Complex& secondComplex) const;
  Complex divide(const Complex& secondComplex) const;
  double abs() const;
  string toString() const;

  Complex& operator+=(Complex& secondComplex);
  Complex& operator-=(Complex& secondComplex);
  Complex& operator*=(Complex& secondComplex);
  Complex& operator/=(Complex& secondComplex);

  double& operator[](const int&  index);

  Complex& operator++(); // Prefix ++
```

```
  double& operator[](const int&  index);

  Complex& operator++(); // Prefix ++
  Complex& operator--(); // Prefix --

  Complex operator++(int dummy); // Postfix ++
  Complex operator--(int dummy); // Postfix --

  Complex operator+(); // Unary +
  Complex operator-(); // Unary -

  friend ostream& operator<<(ostream& stream, const Complex& complex);
  friend istream& operator >> (istream& stream, Complex& complex);

private:
  double a;
  double b;
};

// Non-member operator functions
Complex operator+(const Complex& c1, const Complex& c2);
Complex operator-(const Complex& c1, const Complex& c2);
Complex operator*(const Complex& c1, const Complex& c2);
```

```
Complex operator/(const Complex& c1, const Complex& c2);

bool operator<(const Complex& c1, const Complex& c2);
bool operator<=(const Complex& c1, const Complex& c2);
bool operator==(const Complex& c1, const Complex& c2);
bool operator!=(const Complex& c1, const Complex& c2);
bool operator>(const Complex& c1, const Complex& c2);
bool operator>=(const Complex& c1, const Complex& c2);


// WRITE YOUR CODE to implement the Complex class here
Complex::Complex(){
    a = 0;
    b = 0;
}
Complex::Complex(double a){
    this->a = a;
    b = 1;
}
Complex::Complex(double a, double b){
    this->a = a;
    this->b = b;
}

double Complex::getRealPart() const{
    return a;
}
double Complex::getImaginaryPart() const{
    return b;
}

Complex Complex::add(const Complex& secondComplex) const{
    double newA = a + secondComplex.getRealPart();
    double newB = b + secondComplex.getImaginaryPart();
    Complex n(newA, newB);
    return n;
}
Complex Complex::subtract(const Complex& secondComplex) const{
    double newA = a - secondComplex.getRealPart();
    double newB = b - secondComplex.getImaginaryPart();
    Complex n(newA, newB);
    return n;
}
Complex Complex::multiply(const Complex& secondComplex) const{

    double c = secondComplex.getRealPart();
    double d = secondComplex.getImaginaryPart();
    double newA =  ((a*c) - (b*d));
    double newB =  ((b*c) + (a*d));
    Complex n(newA, newB);
```

```
    return n;
}
Complex Complex::divide(const Complex& secondComplex) const{

    double c = secondComplex.getRealPart();
    double d = secondComplex.getImaginaryPart();
    double newA =  (((a*c) + (b*d))) / ((c*c) + (d*d));
    double newB =  (((b*c) - (a*d))) / ((c*c) + (d*d));
    Complex n(newA, newB);
    return n;
}
double Complex::abs() const{
    double n = sqrt((a*a) + (b*b));
    return n;
}
string Complex::toString() const{
    string n;
    if(b != 0){
        n = to_string(a) + " + " + to_string(b) + "i";
    }else{
        n = to_string(a);
    }
    return n;
}
Complex& Complex::operator+=(Complex& secondComplex){
    *this = add(secondComplex);
    return *this;
}
Complex& Complex::operator-=(Complex& secondComplex){
    *this = subtract(secondComplex);
    return *this;
}
Complex& Complex::operator*=(Complex& secondComplex){
    *this = multiply(secondComplex);
    return *this;
}
Complex& Complex::operator/=(Complex& secondComplex){
    *this = divide(secondComplex);
    return *this;
}
double& Complex::operator[](const int&  index){
    if(index == 0){
        return a;
    }else{
        return b;
    }
}
Complex& Complex::operator++(){
    a += 1;
    //Complex n(newA, b);
```

```
    return *this;
}
Complex& Complex::operator--(){
    a -= 1;
    //Complex n(a, b);
    return *this;
}
Complex Complex::operator++(int dummy){
    Complex n(a, b);
    a += 1;
    return n;
}
Complex Complex::operator--(int dummy){
    Complex n(a, b);
    a -= 1;
    return n;
}
Complex Complex::operator+(){
    return *this;
}
Complex Complex::operator-(){
    return Complex (-a, -b);
}
ostream& operator<<(ostream& stream, const Complex& complex){
    if(complex.b == 0){
        stream<<complex.a;
    }else{
        stream<<complex.a<<" + "<<complex.b<<"i";
    }
    return stream;
}
istream& operator>>(istream& stream, Complex& complex){
    cout<<"Enter a: ";
    stream>>complex.a;
    cout<<"Enter b: ";
    stream>>complex.b;

    return stream;
}
Complex operator+(const Complex& c1, const Complex& c2){
    return c1.add(c2);
}
Complex operator-(const Complex& c1, const Complex& c2){
    return c1.subtract(c2);
}
Complex operator*(const Complex& c1, const Complex& c2){
    return c1.multiply(c2);
}
Complex operator/(const Complex& c1, const Complex& c2){
    return c1.divide(c2);
```

```
}
bool operator<(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() < c2.getRealPart());
}
bool operator<=(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() <= c2.getRealPart());
}
bool operator==(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() == c2.getRealPart());
}
bool operator!=(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() != c2.getRealPart());
}
bool operator>(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() > c2.getRealPart());
}
bool operator>=(const Complex& c1, const Complex& c2){
    return (c1.getRealPart() >= c2.getRealPart());
}


int main()
{
  Complex number1;
  cout << "Enter the first complex number: ";
  cin >> number1;

  Complex number2;
  cout << "Enter the second complex number: ";
  cin >> number2;

  cout << "(" << number1 << ")" << " + " << "(" << number2
    << ") = " << (number1 + number2) << endl;
  cout << "(" << number1 << ")" << " - " << "(" << number2
    << ") = " << (number1 - number2) << endl;
  cout << "(" << number1 << ")" << " * " << "(" << number2
    << ") = " << (number1 * number2) << endl;
  cout << "(" << number1 << ")" << " / " << "(" << number2
    << ") = " << (number1 / number2) << endl;
  cout << "|" << number1 << "|" << " = " << number1.abs() << endl;
  cout << "|" << number2 << "|" << " = " << number2.abs() << endl;

  number1[0] = 3.4;
  cout << number1++ << endl;
  cout << ++number2 << endl;
  cout << (3 + number2) << endl;
  cout << (number2 += number1) << endl;
  cout << (number2 *= number1) << endl;

  number1 = number2;
```

```
  cout << number1 << endl;
  cout << number2 << endl;

  cout << (number1 < number2) << endl;
  cout << (number1 <= number2) << endl;
  cout << (number1 == number2) << endl;
  cout << (number1 != number2) << endl;
  cout << (number1 > number2) << endl;
  cout << (number1 >= number2) << endl;

  return 0;
}
```

14.11 (*Algebra: vertex form equations*) The equation of a parabola can be expressed in either standard form ($y = ax^2 + bx + c = 0$) or vertex form ($y = a(x - h)^2 + k$). Write a program that prompts the user to enter a, b, and c as integers in standard form and displays $h\left(=\frac{-b}{2a}\right)$ and $k\left(=\frac{4ac-b^2}{4a}\right)$ in the vertex form. Use the template in **https://liangcpp.pearsoncmg.com/test/Exercise14_11.txt to write your code.

Exercise14_11 Template

Sample Run for Exercise14_11.cpp

Enter input data for the program (Sample data provided below. You may modify it.)

```
1 3 1
```

Show the Sample Output Using the Preceeding Input | Reset

Execution Result:

```
command>Exercise14_11
Enter a, b, c: 1 3 1
h is -3/2 k is -5/4

command>
```

```
// Search for "WRITE YOUR CODE" to complete this program
#ifndef RATIONALWITHOPERATORS_H
#define RATIONALWITHOPERATORS_H
#include <string>
#include <iostream>
using namespace std;

class Rational
{
public:
  Rational();
  Rational(int numerator, int denominator);
  int getNumerator() const;
  int getDenominator() const;
  Rational add(const Rational& secondRational) const;
  Rational subtract(const Rational& secondRational) const;
  Rational multiply(const Rational& secondRational) const;
  Rational divide(const Rational& secondRational) const;
  int compareTo(const Rational& secondRational) const;
  bool equals(const Rational& secondRational) const;
  int intValue() const;
  double doubleValue() const;
  string toString() const;
```

```
double doubleValue() const;
string toString() const;

Rational(int numerator); // Suitable for type conversion

// Define function operators for augmented operators
Rational& operator+=(const Rational& secondRational);
Rational& operator-=(const Rational& secondRational);
Rational& operator*=(const Rational& secondRational);
Rational& operator/=(const Rational& secondRational);

// Define function operator []
int& operator[](int index);

// Define function operators for prefix ++ and --
Rational& operator++();
Rational& operator--();

// Define function operators for postfix ++ and --
Rational operator++(int dummy);
Rational operator--(int dummy);

// Define function operators for unary + and -
Rational operator+();
Rational operator-();

// Define the << and >> operators
friend ostream& operator<<(ostream&, const Rational&);
friend istream& operator>>(istream&, Rational&);
```

```
private:
  int numerator;
  int denominator;
  static int gcd(int n, int d);
};

// Define nonmember function operators for relational operators
bool operator<(const Rational& r1, const Rational& r2);
bool operator<=(const Rational& r1, const Rational& r2);
bool operator>(const Rational& r1, const Rational& r2);
bool operator>=(const Rational& r1, const Rational& r2);
bool operator==(const Rational& r1, const Rational& r2);
bool operator!=(const Rational& r1, const Rational& r2);

// Define nonmember function operators for arithmetic operators
Rational operator+(const Rational& r1, const Rational& r2);
Rational operator-(const Rational& r1, const Rational& r2);
Rational operator*(const Rational& r1, const Rational& r2);
Rational operator/(const Rational& r1, const Rational& r2);

#endif

#include <cstdlib> // For the abs function

Rational::Rational()
{
  numerator = 0;
  denominator = 1;
}

Rational::Rational(int numerator, int denominator)
{
  int factor = gcd(numerator, denominator);
  this->numerator = (denominator > 0 ? 1 : -1) * numerator / factor;
  this->denominator = abs(denominator) / factor;
}

int Rational::getNumerator() const
{
  return numerator;
}

int Rational::getDenominator() const
{
  return denominator;
}

// Find GCD of two numbers
int Rational::gcd(int n, int d)
```

```
{
  int n1 = abs(n);
  int n2 = abs(d);
  int gcd = 1;

  for (int k = 1; k <= n1 && k <= n2; k++)
  {
    if (n1 % k == 0 && n2 % k == 0)
      gcd = k;
  }

  return gcd;
}

Rational Rational::add(const Rational& secondRational) const
{
  int n = numerator * secondRational.getDenominator() +
    denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::subtract(const Rational& secondRational) const
{
  int n = numerator * secondRational.getDenominator()
    - denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::multiply(const Rational& secondRational) const
{
  int n = numerator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::divide(const Rational& secondRational) const
{
  int n = numerator * secondRational.getDenominator();
  int d = denominator * secondRational.numerator;
  return Rational(n, d);
}

int Rational::compareTo(const Rational& secondRational) const
{
  Rational temp = subtract(secondRational);
  if (temp.getNumerator() < 0)
    return -1;
  else if (temp.getNumerator() == 0)
```

```
    return 0;
  else
    return 1;
}

bool Rational::equals(const Rational& secondRational) const
{
  if (compareTo(secondRational) == 0)
    return true;
  else
    return false;
}

int Rational::intValue() const
{
  return getNumerator() / getDenominator();
}

double Rational::doubleValue() const
{
  return 1.0 * getNumerator() / getDenominator();
}

string Rational::toString() const
{
  if (denominator == 1)
    return to_string(numerator); // See Ch7 for to_string
  else
    return  to_string(numerator) + "/" + to_string(denominator);
}

Rational::Rational(int numerator) // Suitable for type conversion
{
  this->numerator = numerator;
  this->denominator = 1;
}

// Define function operators for augmented operators
Rational& Rational::operator+=(const Rational& secondRational)
{
  *this = add(secondRational);
  return *this;
}

Rational& Rational::operator-=(const Rational& secondRational)
{
  *this = subtract(secondRational);
  return *this;
}
```

```
Rational& Rational::operator*=(const Rational& secondRational)
{
  *this = multiply(secondRational);
  return *this;
}

Rational& Rational::operator/=(const Rational& secondRational)
{
  *this = divide(secondRational);
  return *this;
}

// Define function operator []
int& Rational::operator[](int index)
{
  if (index == 0)
    return numerator;
  else
    return denominator;
}

// Define function operators for prefix ++ and --
Rational& Rational::operator++()
{
  numerator += denominator;
  return *this;
}

Rational& Rational::operator--()
{
  numerator -= denominator;
  return *this;
}

// Define function operators for postfix ++ and --
Rational Rational::operator++(int dummy)
{
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}

Rational Rational::operator--(int dummy)
{
  Rational temp(numerator, denominator);
  numerator -= denominator;
  return temp;
}

// Define function operators for unary + and -
```

```
Rational Rational::operator+()
{
  return *this;
}

Rational Rational::operator-()
{
  return Rational(-numerator, denominator);
}

// Define the output and input operator
ostream& operator<<(ostream& out, const Rational& rational)
{
  if (rational.denominator == 1)
    out << rational.numerator;
  else
    out << rational.numerator << "/" << rational.denominator;
  return out;
}

istream& operator>>(istream& in, Rational& rational)
{
  cout << "Enter numerator: ";
  in >> rational.numerator;

  cout << "Enter denominator: ";
  in >> rational.denominator;
  return in;
}

// Define function operators for relational operators
bool operator<(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) < 0);
}

bool operator<=(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) <= 0);
}

bool operator>(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) > 0);
}

bool operator>=(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) >= 0);
}
```

```
bool operator==(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) == 0);
}

bool operator!=(const Rational& r1, const Rational& r2)
{
  return (r1.compareTo(r2) != 0);
}

// Define non-member function operators for arithmetic operators
Rational operator+(const Rational& r1, const Rational& r2)
{
  return r1.add(r2);
}

Rational operator-(const Rational& r1, const Rational& r2)
{
  return r1.subtract(r2);
}

Rational operator*(const Rational& r1, const Rational& r2)
{
  return r1.multiply(r2);
}

Rational operator/(const Rational& r1, const Rational& r2)
{
  return r1.divide(r2);
}

int main()
{
  cout<<"Enter a, b, and c"<<endl;
  int a, b, c;
  cin>>a>>b>>c;

  Rational h(-b, (2*a));
  Rational k(((4*a*c)-(b*b)), (4*a));
  cout<<"h is "<<h<<endl;
  cout<<"k is "<<k<<endl;
}
```