自訂有理數 Rational Class

先決定要用甚麼表達方式

```
class Rational {
    int numer;
    int denom;
};
```

上面的兩個 private member 分別用來表示有理數的分子和分母。接下來要寫 constructor，讓 class 知道對於新產生的 object，應該如何設定初始狀態。底下是三種 constructors，有了這三個 constructors，可以產生型別為 Rational 的變數。

```
class Rational
{
    int numer;  // private members
    int denom;
public:
    Rational(): numer{0}, denom{1} {}
    Rational(int n): numer{n}, denom{1} {}
    Rational(int n, int d): numer{n}, denom{d} {}
};
```

例如底下的寫法， a 的值會是 0，b 的值是 3/1，c 的值則是 3/4。

```
int main()
{
    Rational a;
    Rational b{3};
    Rational c{3,4};
}
```

接下來定義 show()，讓我們可以把 Rational 的內容顯示出來。程式碼變成

```
#include <iostream>
using namespace std;
class Rational
{
    int numer;
    int denom;
public:
    Rational(): numer{0}, denom{1} {}
    Rational(int n): numer{n}, denom{1} {}
    Rational(int n, int d): numer{n}, denom{d} {}
    void show() {
        cout << numer << "/" << denom << endl;
    }
};
```

然後在主程式裡面可以這樣用

```
int main()
{
    Rational a;
    Rational b{3};
    Rational c{3,4};
    a.show();
    b.show();
    c.show();
}
```

除了 constructor，我們也要定義 assignment operator。

```
Rational& operator=(const Rational& b)
{
    numer = b.numer;
    denom = b.denom;
    return *this;
}
```

參數定成 const Rational& b ，表示 b 是用 pass by reference 的方式傳參數，b 只是傳進來的參數的代名詞，並沒有複製一份，而是直接參考既有的內容，這時候如果在函數裡面改變 b 的內容，會直接改到原本傳進來的參數 (正版) 的內容，所以，除非真的打算這麼做，不然通常會多加上 const，表示在函數裡面不會動到原本的內容。Assignment operator 的用途是要把等號右邊的內容設定給等號左邊的變數，例如 a = c; 如果當成函數來看相當於 a.=(c)，a 是 Rational 物件，=( ) 就是上面定義的 a 的 operator = 函數，c 是參數，所以在函數裡面，b.numer 和 b.denom 相當於取出 c.numer 和 c.denom。由於 operator= 是 Rational 的成員函數 (member function)，所以可以直接用 . 符號，存取 private 成員 numer 和 denom。
有了上面的 member functions，主程式 main() 裡面可以這樣使用

```
    a = c;
    a.show();
    a = Rational{3,5};
    a.show();
```

把一個 有理數的值設給另外一個有理數。

替 Rational 加入下面的 public member functions (加在 Rational class 的 public: 區段裡面)

```
int nu() const { return numer; }
int de() const { return denom; }
```

在 nu() 和 de() 後面的 const，表示 nu() 和 de() 是 const member functions，不會更改 object 的內容。這兩個函數是為了提供介面，讓 Rational 成員函數以外的其他函數，也能夠取得 private 成員 numer 和 denom 的值。
譬如 operator+，不是 Rational 的成員函數，當我們寫 a + b 的時候，相當於呼叫 +(a,b) 雖然 operator+ 不是 Rational 的成員函數，卻需要用到 a 和 b 兩個 Rational 的 private 內容，這時候就必須透過上面的 nu() 和 de() 來取得 Rational 的 private 成員 numer 和 denom 的值。

```
const Rational operator+(const Rational& lhs, const Rational& rhs)
{
    return Rational(lhs.nu()*rhs.de()+lhs.de()*rhs.nu(),
                    lhs.de()*rhs.de());
}
```

接下來可以把其他 operator 一一定出來。其中 += -= *= /= ++ -- 都是 Rational 的成員函數，而 + - * / 和 < == 這些 binary operator 都不是 Rational 的成員函數。其中 < 和 == 這兩個函數很有用，因為我們可以產生 vector <Rational>，也就是由 Rational 構成的向量，然後如果已經定義了 < 和 ==，就可以利用現成的 sort 函數，對 Rational 向量排序。

底下是稍微完整的 Rational class 的程式碼。
可以特別看一下 << 符號的定義方式，這樣就能取代前面定義的 show()，改用 cout << 的寫法顯示 Rational 的內容。另外也可以看一下 ++ 符號的定義方式，分成 ++a 和 a++ 兩種形式。

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Rational {
    int numer;
    int denom;
    int gcd(int a, int b)
    {
        if (b==0) return a;
        else return gcd(b, a%b);
    }
    int abs(int a)
    {
        return (a>0) ? a : -a;
    }
    void simplify()
    {
        bool negative = (numer*denom < 0);
        numer = abs(numer);
        denom = abs(denom);
        int g = gcd(numer, denom);
        numer = numer/g;
        denom = denom/g;
        if (negative) {
            numer = -numer;
        }
    }
public:
    Rational() { numer=0; denom=1; }
    Rational(int n): numer{n} { denom=1; }
    Rational(int n, int d): numer{n}, denom{d}
    {
        simplify();
    }

    Rational(const Rational & r)
    {
        numer = r.numer;
        denom = r.denom;
    }

    Rational& operator=(const Rational& r)
    {
        numer = r.numer;
        denom = r.denom;
        return *this;
    }

    void show() const
    {
        if (denom==1)
            cout << numer << endl;
        else
            cout << numer << "/" << denom <<
endl;
    }
    int nu() const
    {
        return numer;
    }
    int de() const
    {
        return denom;
    }

    Rational& operator+=(const Rational& r);
    Rational& operator-=(const Rational& r);
    Rational& operator*=(const Rational& r);
    Rational& operator/=(const Rational& r);

    const Rational& operator++();
    const Rational& operator--();
    Rational operator++(int);
    Rational operator--(int);
};


Rational& Rational::operator+=(const
Rational& r)
{
    numer = numer*r.denom+denom*r.numer;
    denom = denom*r.denom;
    simplify();
    return *this;
}
Rational& Rational::operator-=(const
Rational& r)
{
    numer = numer*r.denom-denom*r.numer;
    denom = denom*r.denom;
    simplify();
    return *this;
}
Rational& Rational::operator*=(const
Rational& r)
{
    numer = numer*r.numer;
    denom = denom*r.denom;
    simplify();
    return *this;
}
Rational& Rational::operator/=(const
Rational& r)
{
    numer = numer*r.denom;
    denom = denom*r.numer;
    simplify();
    return *this;
}
```

```
const Rational operator+(const Rational& lhs,
const Rational& rhs)
{
    return
Rational(lhs.nu()*rhs.de()+lhs.de()*rhs.nu(),
                    lhs.de()*rhs.de());
}
const Rational operator-(const Rational& lhs,
const Rational& rhs)
{
    return Rational(lhs.nu()*rhs.de()-
lhs.de()*rhs.nu(),
                    lhs.de()*rhs.de());
}
const Rational operator*(const Rational& lhs,
const Rational& rhs)
{
    return Rational(lhs.nu()*rhs.nu(),
                    lhs.de()*rhs.de());
}
const Rational operator/(const Rational& lhs,
const Rational& rhs)
{
    return Rational(lhs.nu()*rhs.de(),
                    lhs.de()*rhs.nu());
}

bool operator==(const Rational& lhs, const
Rational& rhs)
{
    return lhs.nu()==rhs.nu() &&
                    lhs.de()==rhs.de();
}

bool operator<(const Rational& lhs, const
Rational& rhs)
{
    Rational rat = lhs-rhs;
    return rat.nu()<0;
}

ostream & operator<<(ostream& os, const
Rational& a)
{
    if (a.de()==1)
        os << a.nu() ;
    else
        os << a.nu() << "/" << a.de();
    return os;
}
const Rational& Rational::operator++()
{
    numer += denom;
    return *this;
}
const Rational& Rational::operator--()
{
    numer -= denom;
    return *this;
}
```

```
Rational Rational::operator++(int)
{
   Rational rat = *this;
   ++*this;
   return rat;
}
Rational Rational::operator--(int)
{
   Rational rat = *this;
   --*this;
   return rat;
}
int main()
{
    Rational r{6, -5};
    Rational t{r};
    Rational z;
    r.show();
    t.show();
    z.show();
    z = t;
    z += t;
    z += Rational(2,3);
    z.show();
    z = t+2;
    z.show();
    z = 2/t;
    z.show();
    ++z;
    z.show();
    z--;
    z.show();
    cout << z << endl;

    Rational a{2,3};
    Rational b{5,7};

    // const Rational operator+(const
Rational& a, const Rational& b);
    // the following statement is wrong
because the left-hand-side is const
    // a+b = 3;

    vector<Rational> vec{
        Rational(2,3),
        Rational(3,4),
        Rational(1,2)};

    cout << "before sorting\n";
    for (auto v: vec)
        cout << v << endl;

    sort(vec.begin(), vec.end());

    cout << "after sorting\n";
    for (auto v: vec)
        cout << v << endl;

}
```