

Objectives

1. To implement an abstract data type (ADT)¹
2. To practice using the operator overloading facility of the C++ language
3. To learn about function objects and how to define them

Fraction ADT

Objects: Normalized fractions

Fractions of the form $\frac{a}{b}$ where both a and b are integers, with $b \neq 0$. The integers a and b are called the *numerator* and the *denominator* of the fraction, respectively. Normalized fractions have positive denominators; that is, $b > 0$.

Arithmetic operations

$$\text{Addition: } \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

$$\text{Subtraction: } \frac{a}{b} - \frac{c}{d} = \frac{ad - cb}{bd}$$

$$\text{Multiplication: } \frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

$$\text{Devision: } \frac{a}{b} / \frac{c}{d} = \frac{ad}{bc}$$

$$\text{Negation: } -\frac{a}{b} = \frac{-a}{b}$$

$$\text{Inversion: } \text{inverse}\left(\frac{a}{b}\right) = \frac{b}{a}, \quad a \neq 0$$

$$\text{Normalization: } \text{normalize}\left(\frac{a}{b}\right) = \frac{-a}{-b} \text{ if } b < 0$$

$$\text{Reduction: } \text{reduce}\left(\frac{a}{b}\right) = \frac{a'}{b'} \text{ where} \\ a = ga', b = gb', g = \gcd(a, b)$$

Relational operations

$$\frac{a}{b} = \frac{c}{d} \text{ if and only if } ad = bc$$

$$\frac{a}{b} \neq \frac{c}{d} \text{ if and only if } !\left(\frac{a}{b} = \frac{c}{d}\right)$$

$$\frac{a}{b} < \frac{c}{d} \text{ if and only if } ad < bc$$

$$\frac{a}{b} > \frac{c}{d} \text{ if and only if } \frac{c}{d} < \frac{a}{b}$$

$$\frac{a}{b} \leq \frac{c}{d} \text{ if and only if } !\left(\frac{a}{b} > \frac{c}{d}\right)$$

$$\frac{a}{b} \geq \frac{c}{d} \text{ if and only if } !\left(\frac{a}{b} < \frac{c}{d}\right)$$

The fact that the denominators of fractions are positive leads to simple definitions of the relational operations above.

¹ADT = Objects + Operations – Implementation details

A fraction is said to be in *normalized* form if its denominator is positive. For example, the fraction $\frac{3}{-5}$ is not normalized but can be normalized to $\frac{-3}{5}$ where the numerator absorbs and represents the sign of the fraction.

A fraction is said to be *reduced* to lowest terms if its numerator and denominator have no common factors other than ± 1 . Thus any fraction can be reduced to lowest terms by dividing both its numerator and denominator by their greatest common divisor (*gcd*). For example, since $5 = \text{gcd}(15, 20)$, the fraction $\frac{15}{20}$ reduces to the fraction $\frac{3}{4}$.

Your Task

Implement a **Fraction** class so that **Fraction** objects can be used as operands in integer arithmetic expressions.

Example 1: mixing integers with Fraction objects

```
1 Fraction_2016 f1; // = (0/1) = 0
2 Fraction_2016 f2(5); // (5/1) = 5
3 Fraction_2016 f3(55, -77); // = (-5/7), both normalized and reduced
4
5 // mixing ints and Fractions
6 Fraction_2016 expr = f1 + 1 - f2 * 2 + (-f3) / 1 + 10;
7 cout << "expr: " << expr << endl;
8 assert(expr == Fraction_2016(12, 7));
9 cout << "f1: " << f1 << endl;
10 cout << "f2: " << f2 << endl;
11 cout << "f3: " << f3 << endl;
12 cout << "victory!" << endl;
```

Output

```
13 expr: 12/7 = 1 + 5/7
14 f1: 0
15 f2: 5
16 f3: -5/7
17 victory!
```

Notice that class **Fraction** provides three ways to construct **Fraction** objects, as shown on lines 1-3. Class **Fraction** accomplishes the three constructions through a single constructor with two optional parameters of type **long**, using **0** and **1** as the default parameter values for the numerator and denominator, respectively.

Lines 3 and 16 show that the fraction under construction is both normalized and reduced. The supplied numerator and denominator values must be adjusted into a normalized fraction so that it may later be used in a relational expression—recall that the definitions of relational operators presented on page 1 applies only to normalized fractions.

The reduction part of a newly constructed fraction is an attempt to minimize unnecessary arithmetic overflow as the values of the numerator and denominator can grow in size very rapidly:

An example of growth rate of numerator and denominator

$$\text{without reduction} \quad \frac{1}{111} + \frac{1}{222} + \frac{1}{333} = \frac{333}{24642} + \frac{1}{333} = \frac{135531}{8205786}$$

$$\text{with reduction} \quad \frac{1}{111} + \frac{1}{222} + \frac{1}{333} = \frac{333}{24642} + \frac{1}{333} = \frac{1}{74} + \frac{1}{333} = \frac{407}{24642} = \frac{11}{666}$$

Note that $\frac{135531}{8205786} = \frac{12321}{12321} \times \frac{11}{666}$

Thus, **Fraction** applies reduction whenever possible during an operation.

The expression on line 6 necessitates operator overloads that can perform operations of the forms $-f$, $f * i$, f/i , $f + i$ and $f_1 - f_2$, where f , f_1 and f_2 denote **Fraction** objects and i denotes an **int**.

Line 7 requires an operator overload that can write a **Fraction** object to **cout**, and line 13 shows how it should look like when the numerator is \geq the denominator. Lines 14-15 show how a fraction like **f2** whose denominator is **1** should be written to **cout**: without the **"/1"** part.

Relative to your previous assignments, this assignment is easy and involves no dynamic storage management. It involves implementing about 40 operator overloads, most of which are one-liner functions. The **Fraction** class itself is rather simple, containing a pair of **long** data members for storing the numerator and denominator, a pair of accessors (getters) and mutators (setters) for each data member, and a single constructor with two optional parameters.

To be clear about which 40 operator overloads to implement, Tables 1 and 2 on pages 5 and 6, respectively, list the prototypes for the required member and non-member operator overloads. As a result, you save time and effort with figuring out the return and parameter types and with deciding whether to implement an operator overload as a member of the class or as a non-member function.

Although it is possible to turn objects of a class into function objects in many ways by overloading the function call operator **operator()** as many times as desired, **Fraction** objects are turned into function objects in only one way, as prototyped on Table 1 at line 16.

For example, for a **Fraction** object **f** that stores the fraction $-\frac{5}{7}$ the call **f()** should return the string "**(-5/7)**".

Table 1, line 17, prototypes the dereference operator **operator***, enabling **Fraction** objects to behave like pointers, although **Fraction** objects have nothing to do with pointers! For example, for object **f** from the above paragraph, the expression ***f** really is **f** itself, behaving like (but only pretending to be) a pointer!

Please bear in mind that the dereference and function call operators and indeed most operators do not necessarily have to be overloaded for a class unless they provide meaningful service. For example, does it make sense to overload the subscript **operator[]** for **Fraction** objects? Regardless of an answer, one can't afford to miss the opportunity here. In fact, you are going to overload **operator[]** in a way that can be useful but is not too common: rather than implementing the following common overloads

Common Prototypes for Operator[] Overloads

```
// assume that an even k signals a request for the numerator and an odd k for the denominator
long& operator[](int k);
const long& operator[](int k) const;
```

you are going to implement the subscript overloads prototyped as follows:

Less Common Prototypes for Operator[] Overloads

```
// [s] returns the numerator if s="top"; otherwise, returns denominator
long& operator[](const string &s);
const long& operator[](const string &s) const;
```

The idea here is to remind you the although **operator[]** takes a single argument, the argument itself is not limited to type **int** only and can in fact be of any type, such as **string**, for example. To see these operator overloads in action, see lines 250-260 on page 12.

Finally, this assignment provides a rather detailed test driver code which, for reference, is printed on pages 7-13.

Table 1: Member Operator Overload Prototypes

```

1 Fraction& operator= (const int& val); // f = i
2 Fraction& operator+= (const int& val); // f += i
3 Fraction& operator-= (const int& val); // f -= i
4 Fraction& operator*= (const int& val); // f *= i
5 Fraction& operator/= (const int& val); // f /= i
6
7 Fraction& operator= (const Fraction& rhs); // f = f
8 Fraction& operator+= (const Fraction& rhs); // f += f
9 Fraction& operator-= (const Fraction& rhs); // f -= f
10 Fraction& operator*= (const Fraction& rhs); // f *= f
11 Fraction& operator/= (const Fraction& rhs); // f /= f
12
13 Fraction & Fraction::operator++(); // ++f
14 Fraction & Fraction::operator--(); // --f
15
16 string operator()() const; // returns the string version of this fraction
17 Fraction& operator*(); // make Fraction objects behave like pointers: returns *this
18
19 // [s] returns the numerator if s="top"; otherwise, returns denominator
20 long& operator[](const string &s);
21 const long& operator[](const string &s) const;
22
23 // The following operators are usually overloaded as members
24 Fraction operator+(const Fraction& rhs) const; // +f
25 Fraction operator-(const Fraction& rhs) const; // -f
26
27 Fraction Fraction::operator++(int); // f++
28 Fraction Fraction::operator--(int); // f--

```

Table 2: Non-Member Operator Overload Prototypes

```

29 Fraction operator+(const Fraction& lhs, const Fraction& rhs); // f + f
30 Fraction operator+(const Fraction& lhs, const int& val); // f + i
31 Fraction operator+(const int& val, const Fraction& rhs); // i + f
32
33 Fraction operator-(const Fraction& lhs, const Fraction& rhs); // f - f
34 Fraction operator-(const Fraction& lhs, const int& val); // f - i
35 Fraction operator-(const int& val, const Fraction& rhs); // i - f
36
37 Fraction operator*(const Fraction& lhs, const Fraction& rhs); // f * f
38 Fraction operator*(const Fraction& lhs, const int& val); // f * i
39 Fraction operator*(const int& val, const Fraction& rhs); // i * f
40
41 Fraction operator/(const Fraction& lhs, const Fraction& rhs); // f / f
42 Fraction operator/(const Fraction& lhs, const int& val); // f / i
43 Fraction operator/(const int& val, const Fraction& rhs); // i / f
44
45 bool operator==(const Fraction& lhs, const Fraction& rhs); // f == f
46 bool operator==(const Fraction& lhs, const int& val); // f == i
47 bool operator==(const int& val, const Fraction& rhs); // i == f
48 bool operator!=(const Fraction& lhs, const Fraction& rhs); // f != f
49 bool operator!=(const Fraction& lhs, const int& val); // f != i
50 bool operator!=(const int& val, const Fraction& rhs); // i != f
51 // ditto for operators <, <=, >, >=
52
53 istream& operator>> (istream& istr, Fraction& rhs);
54 ostream& operator<< (ostream& ostr, const Fraction& rhs);

```

Test Drive

Test Driver Code: page 1/7

```
1 #include <iostream>
2 #include <iomanip>
3 #include <string>
4 #include <cassert>
5 using namespace std;
6 #include "Fraction.h"
7
8 using namespace std;
9
10 int main()
11 {
12     cout << "Test Fraction and Fractional Computations\n";
13     cout << "-----\n\n";
14
15     cout << "testing default ctor with Fraction f0;\n";
16     Fraction f0;           // test default ctor
17
18     cout << "testing fraction == integer with f0 == 0\n";
19     int zero = 0;
20     assert(f0 == zero);    // test fraction == int
21     assert(f0 == 0);       // test fraction == int literal
22     cout << "f0: " << f0 << '\n';
23     cout << "ok\n\n";
24
25     cout << "testing 1-arg ctor with Fraction f1(5);\n";
26     Fraction f1(5);        // test 1 arg ctor
27     assert(5 == f1);       // test integer == fraction
28     cout << "f1: " << f1 << "\n\n";
29     cout << "ok\n\n";
30
31     cout << "testing copy ctor with Fraction f2 = f1;"
32         << '\n';
33     Fraction f2 = f1;      // test ctor
34     assert(f2 == f1);     // test fraction == fraction
35     cout << "f2: " << f2 << "\n\n";
36     cout << "ok\n\n";
37
38     cout << "testing fraction == integer with f1 == 5;\n";
39     assert(f1 == 5);      // test fraction == integer
40     cout << "ok\n\n";
```

Test Driver Code: page 2/7

```
41
42 cout << "testing fraction == fraction with f1 == f2;\n";
43 assert(f1 == f2);    // test fraction == fraction
44 cout << "ok\n\n";
45
46 cout << "testing fraction != fraction with !(f1 != f2);\n";
47 assert(!(f1 != f2));    // test fraction != fraction
48 cout << "ok\n\n";
49
50 cout << "testing 2 args ctor with Fraction half = "
51      << "Fraction(1, 2);\n";
52 Fraction half = Fraction(1, 2);    // 2 args ctor
53 cout << "half: " << half << "\n\n";
54
55 cout << "testing operator+ with f2 = f1 + half;\n";
56 f2 = f1 + half;    // test operator+
57 cout << "f2: " << f2 << "\n\n";
58
59 cout << "testing operator< with f1 < f2;\n";
60 assert(f1 < f2);    // operator <
61 cout << "ok\n\n";
62
63 cout << "testing operator<= with f1 <= f2;\n";
64 assert(f1 <= f2);    // operator <=
65 cout << "ok\n\n";
66
67 cout << "testing operator> with f2 > f1;\n";
68 assert(f2 > f1);    // operator >
69 cout << "ok\n\n";
70
71 cout << "testing operator>= with f2 >= f1;\n";
72 assert(f2 >= f1);    // operator >=
73 cout << "ok\n\n";
74
75 cout << "testing operator!= with f2 != f1;\n";
76 assert(f2 != f1);    // operator !=
77 cout << "ok\n\n";
78
79 cout << "testing operator==, operator- with "
80      << "f1 == f2 - half;\n";
81 assert(f1 == f2 - half);    // operator -
82 cout << "ok\n\n";
```


Test Driver Code: page 3/7

```
83
84 cout << "testing 2 args ctor with Fraction oneThird"
85      " (1, 3);\n";
86 Fraction oneThird(1, 3);
87 cout << "oneThird: " << oneThird << "\n\n";
88
89 cout << "testing assignment=, binary +, -, and unary - "
90      "with f2 = f1 + oneThird - ( - oneThird );\n";
91 f2 = f1 + oneThird - (-oneThird);
92 cout << "f2: " << f2 << '\n';
93 assert(f2 == Fraction(17, 3));
94 cout << "ok\n\n";
95
96 cout << "testing fractional expression with "
97      "f2 = f1 - oneThird + ( - oneThird );\n";
98 f2 = f1 - oneThird + (-oneThird); // unary +
99 cout << "f2 *: " << f2 << '\n';
100 assert(f2 == Fraction(13, 3));
101 cout << "ok\n\n";
102
103
104 cout << "testing post++ with f2 = f1++;\n";
105 f2 = f1++;
106 cout << "f1 : " << f1 << '\n';
107 cout << "f2 : " << f2 << '\n';
108 assert(f1 == Fraction(6));
109 assert(f2 == Fraction(5));
110 cout << "ok\n\n";
111
112 cout << "testing pre++ with f2 = ++f1;\n";
113 f2 = ++f1;
114 cout << "f1 : " << f1 << '\n';
115 cout << "f2 : " << f2 << '\n';
116 assert(f1 == Fraction(7));
117 assert(f2 == Fraction(7));
118 cout << "ok\n\n";
119
120 cout << "testing post-- with f2 = f1--;\n";
121 f2 = f1--;
122 cout << "f1 : " << f1 << '\n';
123 cout << "f2 : " << f2 << '\n';
124 assert(f1 == Fraction(6));
125 assert(f2 == Fraction(7));
126 cout << "ok\n\n";
```

Test Driver Code: page 4/7

```
127
128 cout << "testing pre-- with f2 = --f1;\n";
129 f2 = --f1;
130 cout << "f1 : " << f1 << '\n';
131 cout << "f2 : " << f2 << '\n';
132 assert(f1 == Fraction(5));
133 assert(f2 == Fraction(5));
134 cout << "ok\n\n";
135
136 f1 = Fraction(6) / 10;
137 cout << "testing normalization with f1 == \"6/10\";\n";
138 assert(f1 == Fraction(3, 5));
139 cout << "f1 : " << f1 << '\n';
140 cout << "ok\n\n";
141
142 cout << "computing sum = 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + "
143         " 1/3 + 1/3 + 1/3 + 1/3 + 1/3\n";
144 Fraction sum;          // sum = 0
145 for (int k = 0; k < 10; ++k)
146 {
147     sum = sum + oneThird;
148     cout << "sum: " << sum << '\n';
149 }
150 cout << "sum: " << sum << '\n';
151 assert(sum == Fraction(10, 3));
152 cout << "ok\n\n";
153
154 cout << "testing operators * and / with "
155         "sum = sum * oneThird / oneThird;\n";
156 sum = sum * oneThird / oneThird;
157 cout << "sum: " << sum << '\n';
158 assert(sum == Fraction(10, 3));
159 cout << "ok\n\n";
160
161 cout << "computing (1/2) + (-1/3) + (1/4) + (-1/5) + "
162         "(1/6) + (-1/7) + (1/8) + (-1/9)" << "\n\n";
163 cout << setw(5) << 'k' << setw(15) << 'd' << " : " << 's' << '\n';
164 cout << setw(5) << '-' << setw(15) << '-' << " : " << '-' << '\n';
165 Fraction sum_alternate, sum_even, sum_odd;
166 double d = 0.0;
167 int one = 1;
168 for (int k = 2; k < 10; ++k)
169 {
170     Fraction f3(1, k);
171     if(k%2 == 0) sum_even += f3;
172     else         sum_odd  += f3;
```

Test Driver Code: page 5/7

```
173
174 sum_alternate = sum_alternate + Fraction(one, k);
175 d = d + static_cast<double>(one) / k;
176 one = -one;
177 cout << setw(5) << k << setw(15) << d << " : " << sum_alternate << '\n';
178
179 }
180 assert(sum_alternate == sum_even - sum_odd);
181 cout << "ok\n\n";
182
183 cout << "testing operator>> with cin >> f;\n";
184 Fraction f;
185 cin >> f;
186 cout << "f : " << f << '\n';
187 f = f + Fraction(6, 10) - f;
188
189 cout << "\ntesting operator +=\n";
190 f += 1;
191 cout << "f : " << f << '\n';
192 assert(f == Fraction(8, 5));
193 cout << "f : " << f << '\n';
194 cout << "ok\n\n";
195
196 cout << "testing operator -=\n";
197 f -= Fraction(1);
198 cout << "f : " << f << '\n';
199 assert(f == Fraction(3, 5));
200 cout << "ok\n\n";
201
202 cout << "testing operator *=\n";
203 f *= Fraction(2, 7);
204 cout << "f : " << f << '\n';
205 assert(f == Fraction(6, 35));
206 cout << "ok\n\n";
207
208 cout << "testing operator /=\n";
209 f /= Fraction(3, 5);
210 cout << "f : " << f << '\n';
211 assert(f == Fraction(2, 7));
212 cout << "ok\n\n";
213
214 cout << "testing Fraction + int\n";
215 f = f + 1;
216 cout << "f : " << f << '\n';
217 assert(f == Fraction(9, 7));
218 cout << "ok\n\n";
```

Test Driver Code: page 6/7

```
219
220 cout << "testing Fraction - int\n";
221 f = f - 1;
222 cout << "f : " << f << '\n';
223 assert(f == Fraction(2, 7));
224 cout << "ok\n\n";
225
226
227 cout << "testing int + Fraction\n";
228 f = 1 + f;
229 cout << "f : " << f << '\n';
230 assert(f == Fraction(9, 7));
231 cout << "ok\n\n";
232
233 cout << "testing int - Fraction\n";
234 f = 1 - f;
235 cout << "f : " << f << '\n';
236 assert(f == Fraction(-2, 7));
237 cout << "ok\n\n";
238
239 cout << "testing pointer behavior of f\n";
240 *f = Fraction(11, 22);
241 cout << "f : " << f << '\n';
242 assert(f == Fraction(1, 2));
243 cout << "ok\n\n";
244
245 cout << "testing string version of f\n";
246 cout << "f() : " << f() << '\n';
247 assert(f == Fraction(1, 2));
248 cout << "ok\n\n";
249
250 cout << "testing indexed Fraction objects\n";
251 f["top"] = 77; // same as nominator=77, but no normalization nor reduction
252 f["bottom"] = -154; // same as denominator=-154, but no normalization nor reduction
253 cout << "f : " << f << '\n';
254 assert(f["top"] == 77 && f["bottom"] == -154);
255 cout << "ok\n\n";
256
257 cout << "testing normalization\n";
258 f.normalize();
259 cout << "f : " << f << '\n';
260 assert(f["top"] == -77 && f["bottom"] == 154);
261 cout << "ok\n\n";
```

Test Driver Code: page 7/7

```
262
263 cout << "testing reduction to lowest form\n";
264 f.reduce();
265 cout << "f : " << f << '\n';
266 assert(f["top"] == -1 && f["bottom"] == 2);
267 cout << "ok\n\n";
268
269 cout << "Test completed successfully!" << endl;
270
271 return 0;
272 }
```

Test Drive Output

Output: page 1/4

```
273 Test Fraction and Fractional Computations
274 -----
275
276 testing default ctor with Fraction f0;
277 testing fraction == integer with f0 == 0
278 f0: 0
279 ok
280
281 testing 1-arg ctor with Fraction f1(5);
282 f1: 5
283
284 ok
285
286 testing copy ctor with Fraction f2 = f1;
287 f2: 5
288
289 ok
290
291 testing fraction == integer with f1 == 5;
292 ok
293
294 testing fraction == fraction with f1 == f2;
295 ok
296
297 testing fraction != fraction with !(f1 != f2);
298 ok
299
300 testing 2 args ctor with Fraction half = Fraction(1, 2);
301 half: 1/2
302
303 testing operator+ with f2 = f1 + half;
304 f2: 11/2 = 5 + 1/2
305
306 testing operator< with f1 < f2;
307 ok
308
309 testing operator<= with f1 <= f2;
310 ok
311
312 testing operator> with f2 > f1;
313 ok
```

Output: page 2/4

```
314
315 testing operator>= with f2 >= f1;
316 ok
317
318 testing operator!= with f2 != f1;
319 ok
320
321 testing operator==, operator- with f1 == f2 - half;
322 ok
323
324 testing 2 args ctor with Fraction oneThird (1, 3);
325 oneThird: 1/3
326
327 testing assignment=, binary +, -, and unary - with
328 f2 = f1 + oneThird - ( - oneThird );
329 f2: 17/3 = 5 + 2/3
330 ok
331
332 testing fractional expression with f2 = f1 - oneThird + ( - oneThird );
333 f2 *: 13/3 = 4 + 1/3
334 ok
335
336 testing post++ with f2 = f1++;
337 f1 : 6
338 f2 : 5
339 ok
340
341 testing pre++ with f2 = ++f1;
342 f1 : 7
343 f2 : 7
344 ok
345
346 testing post-- with f2 = f1--;
347 f1 : 6
348 f2 : 7
349 ok
350
351 testing pre-- with f2 = --f1;
352 f1 : 5
353 f2 : 5
354 ok
355
356 testing normalization with f1 == "6/10";
357 f1 : 3/5
358 ok
```

Output: page 3/4

```
358
359 computing sum = 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3
360 sum: 1/3
361 sum: 2/3
362 sum: 1
363 sum: 4/3 = 1 + 1/3
364 sum: 5/3 = 1 + 2/3
365 sum: 2
366 sum: 7/3 = 2 + 1/3
367 sum: 8/3 = 2 + 2/3
368 sum: 3
369 sum: 10/3 = 3 + 1/3
370 sum: 10/3 = 3 + 1/3
371 ok
372
373 testing operators * and / with sum = sum * oneThird / oneThird;
374 sum: 10/3 = 3 + 1/3
375 ok
376
377 computing (1/2) + (-1/3) + (1/4) + (-1/5) + (1/6) + (-1/7) + (1/8) + (-1/9)
378
379      k          d : s
380      -          - : -
381      2          0.5 : 1/2
382      3          0.166667 : 1/6
383      4          0.416667 : 5/12
384      5          0.216667 : 13/60
385      6          0.383333 : 23/60
386      7          0.240476 : 101/420
387      8          0.365476 : 307/840
388      9          0.254365 : 641/2520
389 ok
390
391 testing operator>> with cin >> f;
392
393 numerator? -1
394 denominator? -3
395 f : 1/3
396
397 testing operator +=
398 f : 8/5 = 1 + 3/5
399 f : 8/5 = 1 + 3/5
400 ok
401
402 testing operator -=
403 f : 3/5
404 ok
```


Output: page 4/4

```
405
406 testing operator *=
407 f : 6/35
408 ok
409
410 testing operator /=
411 f : 2/7
412 ok
413
414 testing Fraction + int
415 f : 9/7 = 1 + 2/7
416 ok
417
418 testing Fraction - int
419 f : 2/7
420 ok
421
422 testing int + Fraction
423 f : 9/7 = 1 + 2/7
424 ok
425
426 testing int - Fraction
427 f : -2/7
428 ok
429
430 testing pointer behavior of f
431 f : 1/2
432 ok
433
434 testing string version of f
435 f() : (1/2)
436 ok
437
438 testing indexed Fraction objects
439 f : 77/-154
440 ok
441
442 testing normalization
443 f : -77/154
444 ok
445
446 testing reduction to lowest form
447 f : -1/2
448 ok
449
450 Test completed successfully!
```

Marking scheme

60%	Program correctness
20%	OOP design, Proper use of C++ concepts and facilities, Error checking Simplicity and maintainability of implementation
10%	Format, clarity and completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program