**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИИ**  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.ШУХОВА»**  
**(БГТУ им. В.Г.Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №7

Дисциплина: Обьектно-ориентированное программирование

по теме «Исключительные ситуации в C++»

Выполнил: ст. группы ПВ-21  
Ковалев Павел Александрович

Проверил: Буханов Д.Г.

**Белгород 2020Цель работы**: Получение теоретических знаний об исключительных ситуациях в С++. Получение практических навыков при работе с исключениями в С++.

**Вариант задания 8**

Разработать класс для работы с дробями. Переопределить операции. Сделать возможность работы с неправильными дробями. Предусмотреть исключительные ситуации (деление на ноль, переполнение).

1. #include <QCoreApplication>
2. #include <QTextCodec>
3. *//cout* *<<* *QString::fromUtf8("").toLocal8Bit().data();*
4. #include <cstdio>
5. #include <iostream>
6. #include <exception>
7. #include <limits>
8. #include <cstdlib>
9. #include <string>
10. *using* *namespace* std;
11. *class* **MyException** : exception{
12. *public*:
13. *virtual* *const* char\* ***what***() *const* *throw*() *override* = 0;
14. };
15. *class* **MyOverflowException** : *public* MyException{
16. *private*:
17. long left;
18. long right;
19. *public*:
20. **MyOverflowException**(long a, long b) : MyException(), left(a), right(b){}
21. long **get\_left**() *const* {*return* left;}
22. long **get\_right**() *const* {*return* right;}
23. };
24. *class* **MySumOverflowException** : MyOverflowException{
25. *public*:
26. **MySumOverflowException**(long a, long b) : MyOverflowException(a, b){}
27. *const* char\* ***what***() *const* *throw*(){
    1. char\* str = *new* char[80];
    2. sprintf(str, "Result of %li + %li Overflow\n", get\_left(), get\_right());
    3. *return* str;
28. }
29. };
30. *class* **MyMultOverflowException** : MyOverflowException{
31. *public*:
32. **MyMultOverflowException**(long a, long b) : MyOverflowException(a, b){}
33. *const* char\* ***what***() *const* *throw*(){
    1. char\* str = *new* char[80];
    2. sprintf(str, "Result of %li \* %li Overflow\n", get\_left(), get\_right());
    3. *return* str;
34. }
35. };
36. *class* **MyZeroDivisionException** : MyException{
37. *public*:
38. *const* char\* ***what***() *const* *throw*(){
    1. *return* "MyZeroDivisionError"; }
39. };
40. *class* **Fraction**{
41. *private*:
42. int numerator;
43. unsigned denumerator;
44. void **normalize**();
45. unsigned **gcd**(unsigned, unsigned);
46. unsigned **lcm**(unsigned, unsigned);
47. int **safe\_sum**(int a, int b);
48. int **safe\_mult**(int a, int b);
49. unsigned **safe\_mult**(unsigned a, unsigned b);
50. *public*:
51. **Fraction**() : numerator(0), denumerator(1){}
52. **Fraction**(int num, unsigned denum);
53. **Fraction**(int int\_part, int num, unsigned denum);
54. Fraction *operator*+(*const* Fraction&);
55. Fraction *operator*-(Fraction&);
56. Fraction *operator*\*(*const* Fraction&);
57. Fraction *operator*/(*const* Fraction&);
58. Fraction *operator*-();
59. *friend* ostream& *operator*<< (std::ostream &out, *const* Fraction &f);
60. *friend* istream& *operator*>> (std::istream &in, Fraction &f);
61. };
62. Fraction::**Fraction**(int num, unsigned denum) : numerator(num), denumerator(denum){
63. *if* (denumerator == 0){
64. *throw* (MyZeroDivisionException());
65. }
66. }
67. Fraction::**Fraction**(int int\_part, int num, unsigned denum) : numerator(int\_part \* denum + num), denumerator(denum){
68. *if* (denumerator == 0){
69. *throw* (MyZeroDivisionException());
70. }
71. }
72. unsigned Fraction::**gcd**(unsigned a, unsigned b){
73. *while* (a && b){
74. *if* (a > b) a %= b;
75. *else* b %= a;
76. }
77. *return* a+b;
78. }
79. void Fraction::**normalize**(){
80. *if* (*this*->denumerator == 0){
81. *throw* MyZeroDivisionException();
82. }
83. int a = *this*->numerator;
84. unsigned b = *this*->denumerator;
85. unsigned d = *this*->gcd(abs(a), b);
86. *this*->numerator = a/d;
87. *this*->denumerator = b/d;}
88. int Fraction::**safe\_sum**(int a, int b){
89. long result = a + b;
90. *if* (result > numeric\_limits<int>::max() ||
    1. result < numeric\_limits<int>::min()){
       1. *throw* MySumOverflowException(a, b);
91. }*else*{
92. *return* result;
93. }
94. }
95. int Fraction::**safe\_mult**(int a, int b){
96. long result = a \* b;
97. *if* (result > numeric\_limits<int>::max() ||
    1. result < numeric\_limits<int>::min()){
       1. *throw* MyMultOverflowException(a, b);
98. }*else*{
99. *return* result;
100. }
101. }
102. unsigned Fraction::**safe\_mult**(unsigned a, unsigned b){
103. long unsigned result = a \* b;
104. *if* (result > numeric\_limits<unsigned>::max() ||
     1. result < numeric\_limits<unsigned>::min()){
        1. *throw* MyMultOverflowException(a, b);
105. }*else*{
106. *return* result;
107. }
108. }
109. Fraction Fraction::*operator*-(){
110. *return* Fraction(-*this*->numerator, *this*->denumerator);
111. }
112. Fraction Fraction::*operator*+(*const* Fraction& right){
113. int a\_num = *this*->numerator;
114. int b\_num = right.numerator;
115. unsigned a\_denum = *this*->denumerator;
116. unsigned b\_denum = right.denumerator;
117. unsigned d = gcd(a\_denum, b\_denum);
118. int new\_num;
119. unsigned new\_denum;
120. *try*{
121. new\_denum = *this*->safe\_mult(a\_denum, b\_denum) / d;
122. a\_num \*= new\_denum / a\_denum;
123. b\_num \*= new\_denum / b\_denum;
124. new\_num = safe\_sum(a\_num, b\_num);
125. }
126. *catch*(*const* MyOverflowException &e){
127. std::cout << e.*what*();
128. }
129. Fraction result = Fraction(new\_num, new\_denum);
130. result.normalize();
131. *return* result;}
132. Fraction Fraction::*operator*-(Fraction& right){
133. *return* \**this* + (-right);
134. }
135. Fraction Fraction::*operator*\*(*const* Fraction& right){
136. int a\_num = *this*->numerator;
137. int b\_num = right.numerator;
138. unsigned a\_denum = *this*->denumerator;
139. unsigned b\_denum = right.denumerator;
140. int new\_num;
141. unsigned new\_denum;
142. *try*{
143. new\_num = safe\_mult(a\_num, b\_num);
144. new\_denum = safe\_mult(a\_denum, b\_denum);
145. }
146. *catch*(*const* MyOverflowException &e){
147. cout << e.*what*();
148. }
149. Fraction result = Fraction(new\_num, new\_denum);
150. result.normalize();
151. *return* result;
152. }
153. Fraction Fraction::*operator*/(*const* Fraction& right){
154. Fraction new\_right = Fraction(right.denumerator, right.numerator);
155. *return* \**this* \* new\_right;
156. }
157. std::ostream& *operator*<< (ostream &out, *const* Fraction &f){
158. *if* (abs(f.numerator) >= f.denumerator){
159. int int\_part = f.numerator / (long)f.denumerator;
160. int frac\_part = abs(f.numerator) % (long)f.denumerator;
161. out << int\_part;
162. *if* (frac\_part != 0){
     1. out << ' ' << frac\_part << '/' << f.denumerator;
163. }
164. }*else*{
165. *if* (f.numerator % f.denumerator == 0){
     1. out << f.numerator / f.denumerator;
166. }*else*{
     1. out << f.numerator << "/" << f.denumerator;
167. }
168. }
169. *return* out;
170. }
171. istream& *operator*>> (std::istream &in, Fraction &f){
172. in >> f.numerator;
173. in >> f.denumerator;
174. *try*{
175. f.normalize();
176. }
177. *catch*(*const* MyZeroDivisionException&){
178. cout << QString::fromUtf8("Знаменатель не может быть равен 0").toLocal8Bit().data() << std::endl;
179. }
180. *return* in;
181. }
182. int **main**(int argc, char \*argv[])
183. {
184. QCoreApplication a(*argc*, argv);
185. #ifdef Q\_OS\_WIN32
186. QTextCodec::setCodecForLocale(QTextCodec::codecForName("IBM 866"));
187. #endif
188. #ifdef Q\_OS\_LINUX
189. QTextCodec::setCodecForLocale(QTextCodec::codecForName("UTF-8"));
190. #endif
191. Fraction F(2,3);
192. cout <<F;
193. *return* a.exec();
194. }