Zadanie 1

Zdefiniować klasę **Rect** o prywatnych polach opisujących długości boków (**double**). Zdefiniuj

- konstruktor domyślny, tworzący kwadrat o boku 1;
- konstruktor jednoparametrowy, tworzący kwadrat o podanym boku;
- konstruktor dwuparametrowy (dwa boki);
- metody **getA()** i **getB** zwracające odpowiednie boki prostokąta;
- metodę **getDiagonal()** zwracającą długość przekątnej prostokąta;
- metodę **getArea()** zwracającą pole powierzchni prostokąta;
- metodę **isLargerThan(const Rect&)**, która zwraca **true** gdy *ten* prostokąt ma większe pole od tego przekazanego w argumencie, a **false** w przeciwnym przypadku;
- metodę **info()**, która wypisuje informację o prostokącie, na przykład w formie `Rect[2,3]]` (słowo 'Rect' i w nawiasach kwadratowych długości boków).

Przetestuj wszystkie konstruktory i metody w funkcji **main**.

Zadanie 2

Zdefiniować klasę **Frac** opisującą ułamki (liczby wymierne). W szczególności napisać:

- Konstruktor pobierający licznik `n` i mianownik `d` i tworzący obiekt reprezentujący ułamek $\frac{n}{d}$. Oba argumenty konstruktora powinny mieć wartości domyślne, tak aby
  - obiekt `Frac(n)` odpowiadał liczbie całkowitej `n` (ułamek o mianowniku 1);
  - obiekt `Frac()` reprezentował zero.

  Reprezentacja ułamków powinna być jednoznaczna, czyli niezależnie od wartości argumentów konstruktora powinno być tak, żeby (prywatne) pola num i denom (licznik i mianownik) nie miały wspólnych dzielników (czyli ułamki były zawsze w postaci „skróconej"), denom był zawsze dodatni, a jeśli num jest 0, to denom powinien być 1, aby zapewnić jednoznaczność reprezentacji zera.
  Przyda się pewnie prywatna funkcja znajdująca największy wspólny dzielnik.
  Zauważmy, że klasa ma tylko pola liczbowe, a więc, jak to zwykle w takich przypadkach bywa, nie musimy nadpisywać dostarczonego przez kompilator konstruktora kopiującego.
- Funkcje implementujące działania na ułamkach. Na przykład dla dodawania należy zdefiniować dwie funkcje
  - *metodę* dodającą do *tego* ułamka inny
    ```
    Frac& add(const Frac& f);
    ```
    a więc modyfikującą *ten* obiekt. Metoda powinna zwracać *przez referencję* obiekt, na rzecz którego jest wywoływana;

– *statyczną* funkcję składową
```
static Frac add(const Frac& f1, const Frac& f2);
```
pobierającą przez referencję do stałej dwa obiekty-ułamki i zwracającą przez wartość obiekt reprezentujący sumę tych ułamków (taką funkcję łatwo zaimplementować za pomocą wcześniej zdefiniowanej *metody* z wykorzystaniem konstruktora kopiującego).

i podobnie dla odejmowania, mnożenia, dzielenia.

- Metodę wyprowadzającą na ekran ułamek w formie tekstowej, na przykład dla `Frac(6,-10)` powinniśmy otrzymać napis -3/5.

Tak więc następująca funkcja **main**

download *RatNoOver.cpp*

```
int main() {
      // 2 * ( (2 + 4/10)*5 - 4 ) / (24/15) = 10
    Frac f1 = Frac(2).mul(Frac::sub(Frac(5).mul(
                    Frac::add(Frac(2),Frac(4,10))),
                    Frac(4))).div(Frac(24,15));

        // 7 - 1/3 + (2/6 * 1114) / 111
    Frac f2 = Frac(7).sub(Frac(1,3)).add(
        Frac::div(Frac(2,6).mul(Frac(1114)),Frac(111)));

    f1.show(); std::cout << " ";
    f2.show(); std::cout << std::endl;
}
```

powinna wypisać coś w rodzaju

```
10 3334/333
```

Wykorzystaliśmy tu fakt, że *metody* **add**, **sub**, **mul** i **div** zwracają obiekt **this** przez referencję, więc możliwe jest ich wywoływanie kaskadowe, np.
`frac.add(f1).mul(f2).sub(f3)`
