

# Klasa `fraction`

---

Zadanie polega na zaprojektowaniu, zaimplementowaniu i przetestowaniu klasy `fraction` reprezentującej ułamki, jako całkowite wartości licznika i mianownika.

Zanim rzucimy się do implementacji, trzeba odpowiedzieć na cały szereg pytań. Odpowiedzi na te pytania będą mieć znaczący wpływ na to, jak będzie wyglądać pierwsza wersja utworzonej klasy.

W tym przypadku mamy trochę ułatwione zadanie, bo w samej treści zadania mamy podane wymaganie dotyczące (przynajmniej części) cech obiektów klasy `fraction`.

## Lista pytań

---

Jakie są istotne cechy (atrybuty, stan wewnętrzny) obiektów i jak je reprezentować?

Czy obowiązują jakieś niezmienniki stanu wewnętrznego obiektów?

Które atrybuty można udostępniać publicznie, a które powinny być kontrolowane?

Jak będą tworzone i inicjowane obiekty; czy dopuszczamy istnienie obiektów z nieokreślonym stanem wewnętrznym?

Czy likwidacja obiektu wymaga czynności porządkowych?

Jakie operacje będą wykonywane na obiektach? Które mają być prywatne, które publiczne?

Jakie algorytmy zastosować w operacjach?

Jak program ma korzystać z definicji klasy?

## Decyzje projektowe

---

Atrybuty: `int num, denom`; (licznik, mianownik)

Niezmiennik: `num, denom` względnie pierwsze (nieskracalne);  
przy czym `denom > 0`

Dziedzina: liczby wymierne w zakresie wynikającym z  
reprezentacji typu `int`;

Nadmiar i niedomiar sygnalizowane wyjątkiem `invalid_argument`  
(przyjmijmy na razie, że uwzględniamy tu dzielenie przez 0)

4 sposoby inicjowania obiektów:

- (1) bez argumentów (inicjowanie domyślne);
- (2) przez podanie wartości całkowitej  $\Rightarrow$  `denom == 1`;
- (3) przez podanie pary wartości całkowitych `num, denom`;
- (4) przez podanie innego obiektu `fraction`.

Usuwanie obiektu – bez żadnych czynności porządkowych

# Początki implementacji 1

---

Spróbujmy tym razem zacząć pisanie od testów, w następującym cyklu:

- w pliku zad\_5.cpp zapisuję test nowej funkcjonalności,
- to generuje błędy w testach, a nawet błędy kompilacji,
- naprawiam te błędy implementując nową funkcjonalność.

Cykl powtarzam, aż uznam, że klasa `fraction` ma pełną, przetestowaną funkcjonalność.

Zaczynam od:

```
int main()
{
    fraction f;
    cout << "End of tests.\n";
}
```

## Początki implementacji 1

---

To oczywiście daje błąd kompilacji. Idąc po najmniejszej linii oporu mógłbym wstawić tuż nad `main` definicję klasy (i można to traktować jako słabość tej metody). Zamiast tego dokładam do projektu nagłówek `fraction.h` i włączam go w `zad_5.cpp`. Patrę na inny błąd kompilacji teraz, bo nie napisałem jeszcze samej klasy. Definiuję pustą klasę `fraction` (w nagłówkowym oczywiście) – test dochodzi do szczęśliwego końca.

Teraz mogę w klasie dodać atrybuty (miały być `num` i `denom` typu `int`), konstruktor i metody zwracające wartość licznika i mianownika (detale mam podane parę stron wcześniej). Zaczynam od sprawdzenia w `main`, czy mam ułamek o liczniku 1 i mianowniku 0, a dopiero później uzupełniam definicję klasy.

# Początki implementacji

---

```
class fraction
{
    int num, denom;

public:
    fraction(int num = 0, int denom = 1);
fraction(const fraction& src);

fraction();

    int numerator() const;
    int denominator() const;
}; ←
```

W tym przypadku doskonale sprawdzą się wersje domyślne generowane przez kompilator

; (średnik) kończy definicję klasy!

## Lista inicjalizacyjna w konstruktorze

---

W C++ można inicjować wartości składowych jeszcze zanim rozpocznie się wykonywanie funkcji konstruktora. Przydaje się to szczególnie do inicjowania składowych ustalonych (**const**) – inne przykłady pojawią się później.

W klasie **fraction** nie ma co prawda potrzeby używania listy inicjalizacyjnej, ale dla nabrania wprawy:

```
fraction::fraction(int numerator /* = 0 */,  
                  int denominator /* = 1 */) : num(numerator), denom(denominator)  
{  
    // zostaje normalizacja  
}
```

## Inne ułamki i skracanie

---

Pora dodać kolejny przypadek testowy (powiedzmy, dla  $\frac{1}{2}$ ). Proste skopiowanie pierwszego przypadku skończy się błędem kompilacji, bo `f` jest już zdefiniowane. Jeśli planujemy trochę więcej testów, to za moment polegniemy w morzu zmiennych.

Dodaję zatem blok „ukrywający” poprzedni test i wstawiam sobie kilka par liczników i mianowników.

Do skracania ułamków zdecydowanie przyda się funkcja:

```
int gcd(int m, int n);
```

zdefiniowana poza klasą. Jak każdą inną funkcję trzeba ją przetestować – to znacznie uprości poszukiwanie błędów w innych fragmentach kodu.

Ręce rwą się do skracania, ale warto jeszcze przetestować i zaimplementować operator `==`.



## Skracanie i dalsze testy

---

Mając infrastrukturę gotową testy skracania zajmą ledwie kilka linijek: trzeba w nich uwzględnić narowy użytkownika, który ujemny argument może wstawić i w liczniku, i w mianowniku, a nawet w obu miejscach.

Do zamknięcia testów konstruktora mamy przypadek mianownika równego 0. W tym przypadku trzeba wygenerować wyjątek.

Zanim rzucimy się na testy arytmetyki warto jeszcze napisać i sprawdzić operator  $\ll$ , który pozwoli znacznie skrócić zapis komunikatów o błędach.

I w tym miejscu zaczyna się dość monotonna praca nad kolejnymi operacjami matematycznymi. W dalszej części staram się uzasadnić wybory projektowe.

## Wyobrażenie o działaniu (niezbyt wygodne)

---

Można sobie wyobrazić zdefiniowanie standardowych funkcji składowych do operacji arytmetycznych:

```
fraction lhs(1, 2), rhs(1, 3);  
lhs.Add(rhs);  
lhs.ToStream(cout); cout << endl;  
lhs.Subtract(rhs);  
lhs.ToStream(cout); cout << endl;  
lhs.Divide(rhs);  
lhs.ToStream(cout); cout << endl;  
lhs.Multiply(rhs);  
lhs.ToStream(cout); cout << endl;
```

5/6

1/2

3/2

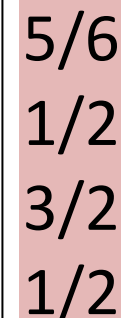
1/2

## Wersja bardziej poręczna z operatorami

---

Wygodniej byłoby jednak:

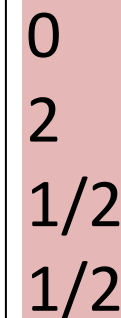
```
cout << (lhs += rhs) << endl;  
cout << (lhs -= rhs) << endl;  
cout << (lhs /= rhs) << endl;  
cout << (lhs *= rhs) << endl;
```



5/6  
1/2  
3/2  
1/2

Warto od razu przyjrzeć się sposobowi wyprowadzania wartości ułamka na strumień wyjściowy:

```
cout << fraction() << endl;  
cout << fraction(2) << endl;  
cout << fraction(1, 2) << endl;  
cout << fraction(7, 14) << endl;
```



0  
2  
1/2  
1/2

Proszę zwrócić uwagę, że operator << nie wyświetla ‘\’ i mianownika, kiedy mianownik jest równy 1.

## Użycie operatorów

---

W zasadzie operatory (funkcje operatorowe) są zmyślną konwencją notacyjną, która pozwala na znacznie wygodniejsze zapisywanie obliczeń. Co prawda w pierwszym odruchu funkcja o nazwie 'operator +=' może budzić pewne opory, ale zdecydowanie wygodniej zapisać:

```
cout << (leftarg += rightarg) << endl;
```

niż:

```
leftarg.Add(rightarg);  
leftarg.ToStream(cout);  
cout << endl;
```

## Test dodawania (+=)

---

```
vector<fraction> lhs{ { 1, 3 }, { 1, 3 }, { 2, 3 }, { 1, 3 }, { -1, 3 }, };
vector<fraction> rhs{ { 0, 1 }, { 1, 3 }, { 1, 3 }, { 5, 3 }, { 1, 3 }, };
vector<fraction> res{ { 1, 3 }, { 2, 3 }, { 1, 1 }, { 2, 1 }, { 0, 1 }, };

for (int i = 0; i < (int)lhs.size(); ++i)
{
    fraction leftarg(lhs[i]);
    leftarg += rhs[i];
    if (leftarg != res[i])
    {
        cout << "Addition error (" << lhs[i] << " + " << rhs[i]
                << ") != " << res[i] << endl;
        err_cnt++;
    }
}
```

Nawet miejsce na komentarz zostało, ale jest chyba zbędny 😊.

## Operator podstawienia

---

Elementem składowym klasy koniecznie musi być operator podstawienia. Nie dość, że modyfikujemy stan obiektu, to jeszcze modyfikowany obiekt musi pojawić się z lewej strony operatora. Deklaracja wygląda następująco:

```
fraction& operator= (const fraction& rhf);
```

Operatory podstawienia zwyczajowo mają wartość typu referencja na obiekt stosownego typu (tu `fraction&`).

Implementacja nie jest skomplikowana:

```
fraction& fraction::operator = (const fraction& rhf)
{
    num = rhf.numerator();
    denom = rhf.denominator();
    return *this;           // czemu bez normalizacji?
}
```

## Operatory podstawienia z operacją

---

O ile czysty operator podstawienia może wygenerować kompilator (można przyjąć, że ta implementacja jest taka sama jak nasza), to pozostałe 4 inne operacje typu podstawienia trzeba zaimplementować samemu:

```
fraction& operator+= (const fraction& rhf);  
fraction& operator-= (const fraction& rhf);  
fraction& operator*= (const fraction& rhf);  
fraction& operator/= (const fraction& rhf);
```

W każdym przypadku wartością funkcji jest referencja na obiekt, na rzecz którego funkcja jest wykonywana (czyli tego z lewej strony operatora). Kończy je instrukcja:

```
return *this;
```

# Operatory arytmetyczne i porównania

---

Standardowe operatory arytmetyczne można definiować w klasie, ale wtedy lewy argument operacji musi być typu

`fraction`. Zatem:

```
fraction left(1, 3);  
fraction right(1, 4);
```

```
cout << left + 1 << endl;  
cout << 1 + right << endl;
```

Pierwsze wyprowadzenie na strumień wyjściowy jest ok, natomiast przy drugim kompilator dał błąd (mnóstwo błędów po prawdzie, a to przez użycie wadliwego wyrażenia przy wyprowadzaniu na strumień).

Promocja typu (`int` do `fraction`) działa po prawej stronie `+` a nie działa po lewej ☹



## Reszta operatorów

---

Prototypy pozostałych operatorów są następujące:

```
fraction operator +(const fraction& lhs, const fraction& rhs);  
fraction operator -(const fraction& lhs, const fraction& rhs);  
fraction operator *(const fraction& lhs, const fraction& rhs);  
fraction operator /(const fraction& lhs, const fraction& rhs);  
fraction operator -(const fraction& rhs);
```

```
bool operator ==(const fraction& lhs, const fraction& rhs);  
bool operator !=(const fraction& lhs, const fraction& rhs);
```

Przy tak zdefiniowanym operatorze dodawania wszystko z poprzedniej strony kompiluje się nienagannie. Czyli + mamy już „symetryczny”, podobnie jak resztę operatorów.

Z rzeczy ważnych: typem funkcji jest `fraction`, bo tworzymy nowy obiekt; funkcje mają dwa argumenty, bo nie ma obiektu, na którego rzecz wołamy funkcję (`this`).

# Wyprowadzanie na strumień wyjściowy

---

Deklaracja tego operatora jest następująca:

```
std::ostream& operator<<(std::ostream& os,  
                        const fraction& rhf);
```

Z jednej strony mamy tu dwa argumenty (bo poza klasami), ale i referencję na strumień jako typ wyjścia (jak w klasie).

To sposób deklarowania funkcji, które modyfikują stan obiektu lewego operandu.

```
std::ostream& operator<<(std::ostream& os, const fraction& rhf)  
{  
    os << rhf.numerator();  
    if (rhf.denominator() != 1)  
        os << '/' << rhf.denominator();  
    return os;  
}
```

## Szersze test klasy `fraction`

---

W zadaniu `fraction` na Moodle znajdziecie plik `zad_5.cpp`, w którym zawarłem swoje testy klasy. Są one podzielone na stosunkowo niezależne od siebie części. Uszeregowanie testów nie jest przypadkowe: najpierw testuję te metody, które przydadzą się w kolejnych testach.

Rozsądnie byłoby sprawdzać klasę `fraction` krok po kroku.

Na początku funkcji `main` jest definicja `TEST_STAGE`.

Ustawiając jej konkretną wartość uruchamiamy początkowe testy (w ogóle jest 19 odcinków testów)..

Kiedy na konsoli zobaczycie radosne: `No errors detected!` można zwiększać wartość (aż do 19). Jeśli nie będzie błędów macie szansę na 5 punktów.

## Dostarczanie rozwiązania

---

Rozwiązaniem zadania są pliki **`fraction.h`** i **`fraction.cpp`**.

Proszę załadować w Moodle tylko te dwa pliki.

Na początku każdego pliku trzeba umieścić w komentarzu imię, nazwisko i nr albumu autora.

Rozwiązanie należy przekazać do:

**1 grudnia 2021 23:59**