Ćwiczenia 10 – strumienie i operatory

Dość istotnym elementem tego ćwiczenia (i kolejnego zadania) będzie operowanie na strumieniach.
Informacje dotyczące tego, jak definiować operatory wyprowadzania i wprowadzania są już znane i nie będą tutaj powtarzane. Główny punkt, to operator wprowadzania danych ze strumienia >> w wersji uproszczonej i nieco bardziej rozbudowanej.
Dodatkowo, do zaimplementowania będą trzy nieco mniej oczywiste operatory, których przykładowe implementacje prześledzimy na przykładzie klasy `fraction`.

Operator jednoargumentowy *

Przykład jest może nieco naciągany, ale wyobraźmy sobie, że nasz odbiorca klasy `fraction` chciałby uzyskać z obiektu `fraction` ułamek właściwy (licznik < mianownik).
Z nieznanych bliżej względów żąda, żeby taka operacja była zaimplementowana jako jednoargumentowy *.
Zatem: typ funkcji to `fraction`, nazwa operator * i zostaje do rozstrzygnięcia, czy powinna to być metoda klasy, czy funkcja zewnętrzna. Ponieważ do implementacji wystarczy znajomość licznika i mianownika, implementujemy poza klasą.

```
fraction operator*(const fraction& rhf)
{
    return fraction(rhf.numerator() % rhf.denominator(),
                    rhf.denominator());
}
```

Operator konwersji typu

Drugi zamówiony przez naszego klienta operator lepiej pasuje do w kontekście jego użycia: chodzi o operator konwersji na wartość całkowitą. Jego wartością jest liczba `int` i chodzi oczywiście o część całkowitą ułamka.
Tutaj, z nazwy operatora (`operator int`) wynika typ wyniku.
Ponieważ operatory konwersji typu nie mają argumentów, musimy ten operator zdefiniować w klasie. Rutynowo takie operatory nie zmieniają wartości obiektu, czego będziemy się trzymać.

```
fraction::operator int() const
{
    return numerator() / denominator();
}
```

Korzystamy z tego operatora „funkcyjnie”: `int(frac)` lub w sposób typowy dla konwersji: `(int)frac` lub naturalnie obiektowo: `frac.operator int()`

Operator postinkrementacji

Ostatni operator „obliczeniowy”, to operator postinkrementacji (`operator ++`). Już sama nazwa zwiastuje problemy, bo jak odróżnić go od operatora preinkrementacji? I czym w ogóle różnią się obie wersje?

Różnica jest w wartości operatora: preinkrementacja zwraca referencję na zmieniony obiekt (a zatem widzimy wartość po zwiększeniu o jeden); postinkrementacja zwraca „starą” wartość obiektu (czyli nie może to być referencja, bo obiekt jest już zmodyfikowany). W obu przypadkach operatory definiujemy zwyczajowo w klasie.

```
fraction fraction::operator++(int)
{
    fraction retv(*this);
    num += denom;
    return retv;
}
```

Ten nienazwany parametr informuje kompilator, że chodzi o post-operator

Operator wprowadzania ze strumienia

Zanim zajmiemy się samym operatorem, trzeba zadbać o to, żeby do testowania operatora nie używać klawiatury – czyli, jak zwykle, automatyzować testy.
Skorzystamy tu z klasy `stringstream` – bardzo poręcznej, bo pozwalającej na użycie strumieni wymiennie z łańcuchami znaków.
Za rozgrzewkę niech posłuży rozbiór sekwencji ułamków na łańcuchy znaków:

```
stringstream ss("7 1/2 7/8");
cout << ss.str() << endl;   // zawartość strumienia na ciąg znaków
while (!ss.eof())           // wszystko przeczytane?
{
    string str;
    ss >> str;              // czytaj do białego znaku
    cout << str << endl;    // wyświetl "wyraz"
}
```

Teraz możemy próbować implementacji operatora bez obawy o połamanie klawiatury.

Operator wprowadzania ze strumienia

Pierwszy krok, czyli odczytanie licznika, nie jest żadnym problemem: wystarczy odczytać wartość całkowitą. Drugi wymaga pewnej gimnastyki, bo mianownik jest (znaczy != 1) ale może go nie być (znaczy == 1). Musimy „spojrzeć” na pierwszy znak stojący za licznikiem w strumieniu. Jeśli jest to `'/'`, możemy czytać mianownik. Problem mamy, kiedy znak jest inny – udało nam się zabrać ze strumienia znak, który do ułamka już nie należy.

Szczęśliwie mamy do dyspozycji metodę `unget()`, która „odkłada” ostatnio odczytany znak do strumienia.

```
char c;
is >> c;
if (c != '/')
    is.unget(); // a co, jeśli się nie powiedzie?
else
    is >> denom;
```

Zadanie 5

Do napisania jest klasa `entry`:

Ma dwie zmienne składowe:

- jedna typu `string` (niech nazywa się `val`)
- druga typu `int` (niech nazywa się `cnt`)

Ma konstruktor z jednym parametrem `const string&`

- ustawia `val` na wartość argumentu, `cnt` na `0`

Ma operator * dający w wyniku wartość `val`

Ma operator `int` dający w wyniku wartość `cnt`

Ma operator postinkrementacji

- wartością funkcji jest „stara” wartość `cnt`
- funkcja zwiększa wartość `cnt` o `1`

Ma operator <

- z argumentem `const entry&`
- porównuje składowe `val`

Zadanie 6

Ma operator << (wyprowadzanie do `ostream`)

z argumentami `ostream&` i `const entry&`
o wartości typu `ostream&`
do strumienia wyjściowego trafia: ‘[‘val’ ‘cnt’]’
np. instrukcja:

```
cout << entry("Yossarian") << endl;
```

powinna na konsoli wyświetlić:

```
[Yossarian 0]
```

Ma operator >> (wprowadzanie z `istream`)

z argumentami `istream&` i `entry&`
o wartości typu `istream&`
czyta w formacie zapisywanym przez <<

Uwaga: dwa ostatnie operatory to funkcje ***zaprzyjaźnione***.

Nieco bardziej złożone wprowadzanie

W podstawowej wersji (tej, która będzie oceniana) operator wprowadzania >> powinien poprawnie obsługiwać wartości jednowyrazowe, tzn. nie zawierające białych znaków:

```
[Yossarian 1]
[Clevinger 2]
```

Wersja za max. 3 punkty

Warto spróbować podejść do obsługi ciągów z pustymi znakami:

```
[Milo Minderbinder 234]
[Major Major Major Major 345]
```

Wersja za max. 4 punkty

Jeśli zostaną Państwu moce przerobowe, rozważcie też:

```
[Catch 22 666]
```

Wersja za max. 5 punktów

Dostarczanie rozwiązania

Rozwiązaniem zadania są pliki **`entry.h`** (definicja klasy i jedno-, dwu-linijkowych metod) i **`entry.cpp`** (definicje dłuższych metod).

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

Rozwiązanie należy załadować w Moodle do:

14 grudnia 2022 23:59