Anagramy

#typ_znakowy #char
#break
#sortowanie #zliczanie
#fill

Anagramy (z greckiego) to słowa (czasem całe zwroty lub zdania) różniące się między sobą tylko kolejnością liter. Na przykład anagramami są słowa **algorytm** i **logarytm**, albo **alfa** i **fala**, ale nie są nimi **lol** i **olo**. Nasuwa się tu skojarzenie ze znaną grą Scrabble: jeśli dwa słowa można ułożyć z tych samych klocków (wykorzystując za każdym razem wszystkie klocki), to te słowa są anagramami.

Napiszemy teraz funkcję `anagram()` sprawdzającą, czy dane dwa wyrazy są anagramami. Aby uprościć sobie życie, będziemy używać tylko małych liter alfabetu łacińskiego (gdzieś już o tym pisaliśmy, chyba w podrozdziale *Palindrom reloaded*.)

Takiej funkcji powinien towarzyszyć program testujący jej działanie: będzie on czytał dwa wyrazy i wypisywał komunikat TAK/NIE. Oto rzeczony program (treść funkcji uzupełnimy za chwilę, nawet w trzech kolejnych podejściach):

```
#include <iostream>
using namespace std;

bool anagram(string a, string b)
{
  . . .
}

int main()
{
  string a, b;
  cin >> a >> b;
  if(anagram(a, b))
    cout << "TAK" << endl;
  else
    cout << "NIE" << endl;
}
```

Podejście pierwsze (praktyczne)

Pierwsza sprawa jest dość prosta: jeśli długości wyrazów są różne, to wtedy nie ma co sprawdzać – od razu zwracamy wartość fałszu. Tak więc pierwszą instrukcją funkcji powinna być instrukcja warunkowa:

```
if(a.length() != b.length())
  return false;
```

Teraz już możemy założyć, że obydwa wyrazy są tej samej długości (będziemy używać wszędzie `a.length()`).

Można by zrobić tak: przespacerujemy się po pierwszym wyrazie (a) i kiedy będziemy przy jego i-tym znaku, wtedy poszukamy tego znaku w drugim wyrazie i tam zmienimy go na jakiś znak, który nie jest literą, na przykład znak kropki (`.`). W ten sposób zaznaczymy, że dany znak już wystąpił. Na koniec tylko sprawdzimy, czy w wyrazie b są same kropki – to będzie sygnał, że składa się on z tych samych klocków Scrabble'a, co wyraz a. Możemy spróbować takiej oto podwójnej pętli (uwaga, to na razie przymiarka):

```
for(int i = 0; i < a.length(); i++)
  for(int j = 0; j < a.length(); j++)
    if(b[j] == a[i])
      b[j] = '.';
```

Zmienna i to numer znaku w wyrazie a, zaś zmienna j to numer znaku w wyrazie b. Znak kropki otaczają pojedyncze apostrofy, bo to typ `char`. Wyglada nieźle, ale czy to działa? Ano *nie bałdzo*, jak w tym sucharze o hrabim.

Załóżmy, że wyraz a to **lol**, a wyraz b to **olo**. Bierzemy znak $a[0]$ (litera **l**) i szukamy go w wyrazie b. OK, jest, i wyraz b wygląda teraz tak: **o.o**. Teraz bierzemy znak $a[1]$ (litera **o**) i szukamy go w b. Jest, ale pasuje w dwóch miejscach! Zatem wyraz b będzie wyglądał tak: `...` – zupełnie, jakby miał być anagramem z wyrazem a. Można temu łatwo zaradzić: wystarczy przerwać pętlę po j, jeśli znajdziemy pasujący znak. Służy do tego instrukcja `break`:

```
for(int i = 0; i < a.length(); i++)
  for(int j = 0; j < a.length(); j++)
    if(b[j] == a[i])
    {
      b[j] = '.';
      break;
    }
```

Należy podkreślić, że tak użyta instrukcja `break` powoduje wyjście tylko z pętli po j, a pętla zewnętrzna (po i) kręci się nadal.

Jeszcze tylko trzeba przespacerować się po wyrazie b i sprawdzić, czy są w nim same kropki. Choć jeden niewykropkowany znak oznacza, że to nie są anagramy. Profesorze de Morgan, widzi Pan to?

Uskładaliśmy już całą (i poprawnie działającą) funkcję `anagram()`:

```
bool anagram(string a, string b)
{
  if(a.length() != b.length())
    return false;
  for(int i = 0; i < a.length(); i++)
```

```
    for(int j = 0; j < a.length(); j++)
      if(b[j] == a[i])
      {
        b[j] = '.';
        break;
      }
  for(int j = 0; j < a.length(); j++)
    if(b[j] != '.')
      return false;
  return true;
}
```

Fajne, ale czy nie da się lepiej? Da się. Po pierwsze: jeśli znaku $a[i]$ nie znajdziemy w drugim wyrazie, wtedy od razu zwracamy wartość `false`. (Spróbujcie napisać taką wersję funkcji `anagram()`.)

Po drugie: da się jeszcze lepiej.

Podejście drugie (sprytne)

Kolejność znaków w obydwu sprawdzanych wyrazach jest raczej przypadkowa. Zgodzicie się chyba, że byłoby nam znacznie łatwiej, gdybyśmy mogli ułożyć znaki w każdym z wyrazów alfabetycznie, wtedy po prostu porównalibyśmy zgodność tych wyrazów *en bloc*.*

Sortowaniem zajmowaliśmy się w poprzednim podrozdziale i poznaliśmy funkcję `sort()`, która na pewno poradziłaby sobie z porządkiem alfabetycznym, tylko jak oznaczyć odkąd dokąd ma ona sortować? Typ danych `string` przypomina tablicę, ale adres jego początku i końca podaje się w zupełnie inny sposób: przy użyciu funkcji `begin()` oraz `end()`.†

Wspominaliśmy już, że typ danych `string` jest typem obiektowym i funkcje (jak znana nam funkcja `length()`) działają na zmienne tego typu przy użyciu operatora kropki. Zatem sortowanie alfabetyczne znaków w wyrazie a może być zrealizowane przy pomocy następującej instrukcji:

```
sort(a.begin(), a.end());
```

Analogicznie sortujemy znaki w wyrazie b.

Teraz trzeba tylko sprawdzić, czy po uporządkowaniu znaków oba wyrazy są równe. Zauważmy, że zdanie logiczne `a==b` ma właśnie taką wartość, jaką powinna zwrócić nasza funkcja, zatem możemy użyć go bezpośrednio w instrukcji `return`. Oto cała funkcja:

```
bool anagram(string a, string b)
{
  sort(a.begin(), a.end());
  sort(b.begin(), b.end());
  return a==b;
}
```

*Znaczy się, w całości.

†Takie funkcje (i kilka innych pokrewnych) zdefiniowane są dla wszystkich kontenerów z biblioteki STL.

Przyznacie, że jej zwięzłość robi wrażenie? A jednak można to zrobić jeszcze lepiej (!) – ze znacznie lepszą złożonością obliczeniową (temat złożoności omówimy kiedy indziej).

Podejście trzecie (optymalne)

Jak to jeszcze ulepszyć? No cóż, jest taki chytry sposób, polegający na zliczeniu ilości wystąpień wszystkich liter w poszczególnych wyrazach. Jeśli to mają być anagramy, to te ilości muszą się zgadzać: tyle samo liter **a**, tyle samo **b** i tak dalej. Klocki Scrabble'a, wszystko jasne. A nad wszystkim czuwać będzie profesor de Morgan...

Liter w alfabecie łacińskim jest dokładnie 26 (od **A** jak piękna Alina, do **Z** jak miła Zosia z Przemyśla), tak więc potrzebujemy dwie tablice liczników, a zatem wybierzemy nazwy $La[\,]$ oraz $Lb[\,]$ o takim właśnie rozmiarze – oddzielnie dla obydwu wyrazów.

```
int La[26], Lb[26];
```

Dobrze byłoby je wyzerować „na dzień dobry", bo przeglądając znaki wyrazów będziemy liczniki powiększać, a zatem ich wartości początkowe są istotne. Można zrobić to jedną prostą pętlą:

```
for(int i = 0; i < 26; i++)
  La[i] = Lb[i] = 0;
```

Zamiast pętlą możemy posłużyć się funkcją `fill()`, która wypełnia zadaną strukturę danych wybraną wartością. Ma ona argumenty takie, jak omawiana w poprzednim podrozdziale funkcja `sort` (czyli określenie „odkąd-dokąd") oraz dodatkowy argument – wartość, która ma być wpisywana. Zerowanie liczników przy pomocy tej funkcji wygląda tak:

```
fill(La, La + 26, 0);
fill(Lb, Lb + 26, 0);
```

Użycie tej funkcji wymaga dołożenia dyrektywy `#include <algorithm>`. Tak naprawdę, to mamy jeszcze inne sposoby wyzerowania tablicy.[‡]

Oczywiście, tym razem przyda się instrukcja warunkowa, którą zastosowaliśmy w podejściu pierwszym – do wykluczenia przypadku różnych długości wyrazów. (W podejściu drugim nie była ona konieczna.)

Początek funkcji `anagram()` wygląda więc tak:

```
if(a.length() != b.length())
  return false;
int La[26], Lb[26];
fill(La, La + 26, 0);
fill(Lb, Lb + 26, 0);
```

[‡]W języku C++ istnieje zgrabny sposób wypełnienia tablicy zerami – bezpośrednio w deklaracji zmiennej tablicowej. Tak powinniśmy zadeklarować nasze liczniki liter: `int La[26] = {0}, Lb[26] = {0};`. Niestety taka sztuczka z wypełnianiem wartością całej tablicy działa tylko dla wartości zero.

Teraz musimy przejść wyrazy od początku do końca – możemy zrobić to w jednej pętli – i powiększać odpowiednie liczniki. Jeśli na przykład jesteśmy przy znaku $a[i]$, musimy powiększyć:

- $La[0]$, jeśli ten znak to **a**,
- $La[1]$, jeśli ten znak to **b**,
- ...,
- $La[25]$, jeśli ten znak to **z**.

Nie ma potrzeby pisania nie wiadomo ilu `if`-ów, gdyż wystarczy wykorzystać fakt, że znaki są w znacznej mierze tożsame z ich kodami ASCII (było, prawda?). Pytanie, czemu równe jest poniższe wyrażenie?

```
a[i] - 'a'
```

Jest to tak naprawdę różnica kodów ASCII tych dwóch znaków: jeśli $a[i]$ jest znakiem **a**, to jest równa zero. Dla znaku **b** jest równa 1, i tak dalej. Dla $a[i]$ będącego znakiem **z** różnica wynosi 25 (dlaczego?).

A zatem jeśli napotkamy znak $a[i]$, wtedy powinniśmy powiększyć licznik La o poniższym numerze:

```
La[a[i] - 'a']++;
```

Pętla zliczająca wystąpienia znaków powinna wyglądać tak:

```
for(int i = 0; i < a.length(); i++)
{
  La[a[i] - 'a']++;
  Lb[b[i] - 'a']++;
}
```

Pozostaje tylko sprawdzić, czy ilości wystąpień poszczególnych liter są takie same (poniżej: pętla po k). Oto kompletna funkcja:

```
bool anagram(string a, string b)
{
  if(a.length() != b.length())
    return false;
  int La[26], Lb[26];
  fill(La, La + 26, 0);
  fill(Lb, Lb + 26, 0);
  for(int i = 0; i < a.length(); i++)
  {
    La[a[i] - 'a']++;
    Lb[b[i] - 'a']++;
```

```
    }
    for(int k = 0; k < 26; k++)
      if(La[k] != Lb[k])
        return false;
    return true;
  }
```

Jest to zdecydowanie najszybszy sposób sprawdzenia, czy wyrazy są anagramami. Dlaczego tak się dzieje? Porozmawiamy o tym w oddzielnym podrozdziale.