Nawiasy

#instrukcja_wyboru
#switch

Doświadczeni programiści wiedzą dobrze, że można programować nie znając do końca danego języka programowania, ale jedna rzecz musi się w programie zgadzać: są to nawiasy!

Poszczególne rodzaje nawiasów muszą być sparowane oraz muszą występować w programie w odpowiedniej kolejności. Oto przykłady poprawnie rozmieszczonych nawiasów (dla prostoty pomijamy inne znaki poza nawiasami i zakładamy, że w kodzie występują trzy rodzaje nawiasów: okrągłe ( ), kwadratowe [ ] oraz klamrowe { }):

```
()()[]
[()[]]
{[]()}
```

A oto przykłady niepoprawne:

```
)(
[)
[(]){}
```

Napiszemy program, który sprawdzi poprawność rozmieszczenia nawiasów w poszczególnych wierszach tekstu. Każdy wiersz będziemy traktować zupełnie oddzielnie i dla każdego wypiszemy komunikat TAK/NIE informujący o tym, czy nawiasy w tym wierszu są poprawne.

Pierwszy wiersz danych wejściowych zawierać będzie dodatnią liczbę całkowitą R oznaczającą ilość wierszy tekstu do przeanalizowania. Każdy kolejny z R wierszy zawierać będzie niepusty ciąg znaków do sprawdzenia, zawierający jedynie znaki nawiasów opisane w treści zadania.

Uchylimy tu rąbka tajemnicy: w rozwiązaniu naszego problemu będzie nam bardzo przydatny stos, który poznaliśmy w poprzednim podrozdziale.

Możemy już powoli naszkicować program – analizą pojedynczego wiersza zajmie się funkcja `row_test()`, którą dopiszemy nieco później:

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

void row_test()
{
  . . .
```

```
}

int main()
{
  int R;
  cin >> R;
  while(R--)
    row_test();
}
```

Funkcja (procedura) `row_test()` powinna oczywiście zacząć od wczytania wiersza tekstu i deklaracji stosu, na którym będziemy odkładać kolejne znaki:

```
string row;
cin >> row;
stack<char> S;
```

Teraz musimy przejść kolejno po wszystkich znakach zmiennej *row* i wykonywać następujące operacje:

- Jeśli kolejnym znakiem ciągu jest nawias otwierający (jakikolwiek), wtedy odkładamy go na stos.
- Jeśli kolejnym znakiem jest nawias zamykający, wtedy sprawdzamy, co mamy na szczycie stosu: jeżeli znak nawiasu ze szczytu pasuje do bieżącego znaku, wtedy zdejmujemy znak ze szczytu stosu i idziemy dalej. Jeśli natomiast na szczycie stosu jest niepasujący nawias lub – o zgrozo! – stos jest pusty, wtedy kicha, nawiasy ustawione są źle i możemy już w tym momencie zakończyć sprawdzanie z negatywnym komunikatem.*

A jak powinien wyglądać stos na zakończenie sprawdzania? Powinien być pusty! Jeśli tak nie jest, to znaczy, że mieliśmy jakiś otwarty nawias (lub ich większą ilość) i nie znaleźliśmy do niego (do nich) zamknięcia.

No fajnie, ale jak to wszystko zapisać? Te dwa wykropkowane punkty wyglądają na jakąś instrukcję warunkową na sterydach i rzeczywiście, mamy w C++ coś takiego, tak zwaną *instrukcję wyboru* (ang. **switch**, wymawiaj: *słicz*).

```
switch(zmienna)
{
  case wartość_1: . . . break;
  case wartość_2: . . . break;
    ⋮
  case wartość_n: . . . break;
  default: . . . break;
}
```

W zależności od wartości, jaką przybiera *zmienna*, wybierany jest wariant kodu. Przed instrukcje `break` można wstawić dowolne zestawy instrukcji. Obecność instrukcji `break` zapewnia, że dla danej wartości wykonywany jest dokładnie określony fragment kodu. Fraza `default`

*Profesorze de Morgan, niezmienny szacunek!

(wymawiaj: *defolt*, z akcentem na *o*) nie musi wystąpić – służy ona do obsługi wszelkich niewymienionych wartości *zmiennej*.

Obsługę różnych wartości można łączyć – i tak właśnie zrobimy w naszym przypadku, gdyż dla każdego otwierającego nawiasu robimy to samo: wstawiamy go na stos. Zatem początek kodu sprawdzającego może wyglądać tak:

```
for(int i = 0; i < row.size(); i++)
  switch(row[i])
  {
    case '(':
    case '[':
    case '{': S.push(row[i]); break;
```

Użyliśmy tutaj funkcji `size()`, ale jest ona tożsama z funkcją `length()`.

A co, gdy *row*[*i*] będzie nawiasem zamykającym, na przykład okrągłym? To już opisaliśmy, i ten wariant kodu powinien wyglądać tak:

```
case ')': if(S.empty() || S.top() != '(')
          {
            cout << "NIE" << endl;
            return;
          }
          else S.pop();
```

Czyli jeśli jest źle, to kończymy funkcję (ale nie cały program), a jeśli dobrze, to zdejmujemy wierzchni element ze stosu. Analogicznie postępujemy z innymi rodzajami nawiasów.[†]

Po zakończeniu pętli należy jeszcze sprawdzić, czy stos jest pusty:

```
if(S.empty())
  cout << "TAK" << endl;
else
  cout << "NIE" << endl;
```

Chyba możemy już przedstawić kompletną funkcję `row_test()`:

```
void row_test()
{
  string row;
  cin >> row;
  stack<char> S;
  for(int i = 0; i < row.size(); i++)
    switch(row[i])
    {
      case '(':
```

[†]Proszę zauważyć, że najpierw sprawdzamy, czy stos jest pusty, bo nie wolno pytać się o wierzchni element pustego stosu.

```
      case '[':
      case '{': S.push(row[i]); break;
      case ')': if(S.empty() || S.top() != '(')
                {
                  cout << "NIE" << endl;
                  return;
                }
                else S.pop();
      case ']': if(S.empty() || S.top() != '[')
                {
                  cout << "NIE" << endl;
                  return;
                }
                else S.pop();
      case '}': if(S.empty() || S.top() != '{')
                {
                  cout << "NIE" << endl;
                  return;
                }
                else S.pop();
    }
  if(S.empty())
    cout << "TAK" << endl;
  else
    cout << "NIE" << endl;
}
```

Pewne fragmenty kodu się powtarzają i można by to zapisać bardziej zwięźle, ale pozostawimy to Czytelnikowi.

Po uruchomieniu programu i wpisaniu przykładowych danych wejściowych:

```
6
()()[]
[()[]]
{[]()}
)(
[)
[(]){}
```

powinniśmy otrzymać poniższy wynik:

```
TAK
TAK
TAK
NIE
NIE
NIE
```