Ilość cyfr

#dzielenie_całkowite
#podłoga #floor
#pętla_do_while
#zmienna_lokalna

Napiszemy teraz program, który znajduje ilość cyfr liczby naturalnej. Na przykład dla liczby 4321 powinniśmy otrzymać wartość 4, natomiast dla liczby 9 powinniśmy otrzymać wartość 1.

Wyznaczeniem ilości cyfr liczby n zajmie się funkcja `digits()` (po angielsku *cyfry* to **digits**, wymawiaj: *dydżits*), której argumentem będzie właśnie liczba n (typu `int`), a rezultatem będzie także liczba naturalna (a więc również typ danych `int`):

```
int digits(int n)
{
  ...
}
```

Jak będziemy zliczać cyfry liczby n? Od lewej czy od prawej? Wbrew pozorom łatwiej zrobić to od prawej, czyli od końca. Przyda się nam pomocnicza zmienna – licznik cyfr, na początku wyzerowany (nazwiemy go `counter`, wymawiaj: *kałnter* – to chyba już było). Będziemy powtarzać następującą sekwencję operacji:

- powiększamy licznik o 1,
- skreślamy ostatnią cyfrę,

aż nam się cyfry skończą. Powiększanie licznika jest proste, to `counter++`, ale jak skreślić ostatnią cyfrę? Prościej niż myślisz: wystarczy n podzielić przez 10. Dzielenie liczb całkowitych odbywa się w C++ zgodnie z regułami dzielenia całkowitego, z odrzuceniem części ułamkowej wyniku. Zatem na przykład 10/4 jest równe 2 (dokładnie), zaś 35/8 jest równe 4. Oznacza to też, że 2/3 jest równe 0, więc trzeba uważać.*

Tak więc jeśli np. liczbę 4321 podzielimy przez 10, otrzymamy 432, a to wygląda tak, jakbyśmy skreślili ostatnią cyfrę. Kolejne dzielenie przez 10 da wynik 43, a jeszcze kolejne: wynik 4. Jeśli teraz podzielimy 4 przez 10 otrzymamy wynik 0 i to jest sygnał zakończenia pętli. Zatem póki $n > 0$, póty pętla powinna się kręcić.

*Takie dzielenie oznacza się w zwykłym zapisie algebraicznym jako $\lfloor \frac{a}{b} \rfloor$ i określa się mianem *podłogi* (ang. **floor**).

Treść funkcji mogłaby wyglądać tak:

```
int counter = 0;
while(n > 0)
{
  counter++;
  n = n / 10;
}
return counter;
```

Uważny czytelnik zorientuje się jednak, że coś jest nie tak. Argumentem funkcji `digits()` ma być liczba naturalna, a więc dopuszczalna jest wartość 0, dla której funkcja powinna zwrócić wartość 1 (zero jest jako żywo liczbą jednocyfrową). Tymczasem nasza funkcja dla $n = 0$ zwraca wartość 0 (sprawdź!).

Można temu zaradzić dodając przed pętlą poniższą instrukcję:[†]

```
if(n == 0)
  return 1;
```

Można jednak postąpić inaczej i będzie to okazja do zaprezentowania kolejnego rodzaju pętli w języku C++, a mianowicie pętli `do-while`. W poznanych do tej pory pętlach `while` oraz `for` sprawdzanie warunku kontynuacji odbywało się *przed* każdym ich obiegiem, co oznacza, że fałszywość warunku powodowała, iż pętla nie obchodziła ani razu. Natomiast w pętli `do-while` sprawdzanie warunku odbywa się *po* każdym obiegu, co gwarantuje, że pętla obejdzie przynajmniej jeden raz.

Ta niewielka z pozoru różnica czyni tę pętlę użyteczną w niektórych sytuacjach, choćby takiej, jaka występuje w naszym problemie. Poniższa pętla da nam pewność, że zmienna `counter` otrzyma poprawną wartość zarówno dla liczby 0, jak i dodatnich liczb naturalnych:

```
int counter = 0;
do
{
  counter++;
  n = n / 10;
}
while(n > 0);
```

Cały program testujący działanie funkcji `digits()` może wyglądać tak:

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

int digits(int n)
{
  int counter = 0;
```

[†]Taka technika nazywana jest powszechnie *wyifowaniem* przypadku.

```
  do
  {
    counter++;
    n = n / 10;
  }
  while(n > 0);
  return counter;
}

int main()
{
  int n;
  cin >> n;
  cout << digits(n) << endl;
}
```

Zaraz, zaraz! Mamy dwa razy zadeklarowaną zmienną n – czy to nie problem? Ależ nie, gdyż jest to zmienna lokalna, której przestrzeń życiowa jest ograniczona do funkcji, w której jest zadeklarowana. Zatem te dwie identycznie nazywające się zmienne są w istocie różnymi bytami. Podczas wywołania `digits(n)` wartość zmiennej n wczytanej dopiero co z klawiatury jest wstawiana jako argument aktualny funkcji.

Podobny numer zrobiliśmy już w podrozdziale *Kot Leonarda* – tam mieliśmy dwukrotnie zadeklarowaną zmienną n w dwóch pętlach `for`. Ciekawe, czy ktoś to zauważył? W tamtym przypadku żywot zmiennej był ograniczony jeszcze bardziej, bo do wnętrza samej pętli. Po co deklarowaliśmy tę zmienną dwukrotnie, skoro można było zadeklarować ją przed pierwszą pętlą i użyć dwukrotnie? Otóż lepiej jest deklarować zmienne jak najbardziej lokalnie, wtedy unika się wielu okazji do popełnienia błędu w konstrukcji programu. Zmienna powinna egzystować i być dostępna tylko wtedy, gdy jest rzeczywiście potrzebna. Niestety, nie zawsze jest to łatwe, a czasem – wręcz niemożliwe.