Kot dziesiętny

#układ_dziesiętny
#schemat_Hornera

Do kompletu brakuje nam jeszcze zamiany liczby z postaci dwójkowej na postać dziesiętną, napiszemy zatem funkcję `decimal()` (wymawiaj: *desimal* z akcentem na *e* i twardym *s*). Tym razem argument funkcji (*s*) będzie typu `string`, natomiast jej rezultat – typu `int`:

```
int decimal(string s)
```

W poprzednim rozdziale poznaliśmy dwójkowy układ pozycyjny i zobaczyliśmy, jak oblicza się wartość dziesiętną liczby – po prostu sumuje się odpowiednie potęgi liczby 2. Warto jednak nauczyć się czegoś nowego, co możemy wykorzystać w innych, bardziej wymagających sytuacjach. Chodzi o schemat Hornera efektywnego obliczania wartości wielomianu.*

Niech będzie dany wielomian stopnia n postaci:

$$W(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0.$$

Możemy go przedstawić w innej, równoważnej formie:

$$W(x) = (\ldots((a_n x + a_{n-1})\, x + a_{n-2})\, x + \cdots + a_1)\, x + a_0$$

Obliczanie wartości $W(x)$ dla danej wartości argumentu x należy rozpocząć od najbardziej wewnętrznego nawiasu: bierzemy a_n, mnożymy przez x, dodajemy a_{n-1}, mnożymy całość przez x, dodajemy a_{n-2}, mnożymy przez x i tak dalej.

W przypadku liczby dwójkowej w roli zmiennej x wystąpi podstawa układu pozycyjnego, czyli liczba 2, zaś kolejne cyfry liczby dwójkowej (*s*) to współczynniki wielomianu: $a_n, a_{n-1}, \ldots, a_1, a_0$. Drobny problem stanowi fakt, że cyfry mamy w postaci znakowej, ale jak już wspominaliśmy w poprzednim podrozdziale, możemy od znaku (*s*[*i*], *i*-ta cyfra) odjąć kod ASCII znaku `'0'`,† co w wyniku da wartość cyfry reprezentowanej przez ten znak.

Proponujemy taką postać całej funkcji:

```
int decimal(string s)
{
  int w{ };
  for(int i{ }; i < s.length(); i++)
    w = w * 2 + s[i] - '0';
  return w;
}
```

*William Horner żył na przełomie XVIII i XIX w., ale ten sposób znany był już wcześniej Newtonowi i matematykom chińskim (tym ostatnim już w XII w.)

†Lub sam znak `'0'`, co na jedno wyjdzie.

Jak widać trick Hornera pozwala na bardzo zwięzły zapis funkcji, a poza tym – co można pokazać – wymaga minimalnej ilości działań arytmetycznych.

Jest jeszcze jedno narzędzie, które możemy teraz wyciągnąć z szuflady: pętlę *for-each* (wymawiaj: *for iicz*), czyli *dla każdego*. Ten typ pętli pojawił się w języku C++ stosunkowo niedawno – wprowadzono go, gdyż doskonale nadaje się do przejrzystego zapisu „spaceru" po elementach struktury danych: tablicy, wektora czy zbioru (`set`). Zamiast podawania zakresu indeksów (numerów) elementów, używa się po prostu kolejnych elementów reprezentowanych przez zmienna o odpowiednim typie. Tak wiec zamiast tradycyjnej pętli przeglądającej string możemy zastosować pętlę z użyciem zmiennej *c* typu znakowego (`char`):

```
int decimal(string s)
{
  int w{ };
  for(char c : s)
    w = w * 2 + c - '0';
  return w;
}
```

Należy to rozumieć: *dla każdego znaku w stringu s wykonaj to czy tamto*. No właśnie: „dla każdego", czyli **for each**.[‡]

Działanie funkcji możemy przetestować przy pomocy przykładowego programu:

```
int main()
{
  string s;
  cin >> s;
  cout << decimal(s) << '\n';
  return 0;
}
```

Na przykład po wpisaniu liczby dwójkowej 11010 otrzymamy wynik 26.

Uzupełnienie programu pozostawiamy Czytelnikowi – jak w poprzednim podrozdziale.

[‡]Tego rodzaju pętla należała już wcześniej do standardu języka Javascript, narzędzia niezwykle użytecznego i powszechnie używanego w tak zwanym *front-endzie*.