Tak dla porządku

#return_0 #string/char[]
#definicja/deklaracja_funkcji

W tym podrozdziale omówimy parę zagadnień, które jakoś ominęliśmy w dotychczasowym tekście, skupiając się na bardziej podstawowych rzeczach.

Instrukcja `return 0` w funkcji `main()`

We wszystkich prezentowanych do tej pory przykładowych programach konsekwentnie zaznaczaliśmy, że funkcja `main()` zwraca wartość całkowitą (`int`), natomiast (chyba) nigdzie nie pojawiła się jawnie instrukcja `return` z wybraną wartością. W przykładach programów, jakie być może Czytelnik widział w internecie, mogła występować instrukcja `return 0` – zwykle jako ostatnia instrukcja programu. Wartość zero zwracana przez funkcję `main()` oznacza, że program zakończył się bez błędu wykonania, czyli na przykład omyłkowego dzielenia przez zero, obliczania pierwiastka kwadratowego z liczby ujemnej czy sięgania do zabronionego obszaru pamięci. Gdyby program zwrócił jakąkolwiek inną wartość, wtedy byłoby to sygnałem, że coś poszło niedobrze, a wtedy system operacyjny otrzymałby informację o błędzie i wyświetliłby odpowiednio obelżywy komunikat. Szczególnie istotne jest to w przypadku korzystania z automatycznej testerki do weryfikacji poprawności rozwiązania zadań algorytmicznych (na przykład Szkopuł czy Codeforces).

Pozwoliliśmy sobie na drobne odstępstwo od standardu, gdyż taki kod wyjściowy programu jest generowany automatycznie i nie musimy zwracać go „ręcznie”. Zawsze to o jedną linijkę w programie mniej.

Czy to znaczy, że możemy w ogóle zapomnieć o `return 0`? Raczej nie, gdyż można wskazać sytuacje, gdy ta instrukcja jest użyteczna – mianowicie wtedy, gdy chcemy przerwać działanie programu w danym miejscu, niekoniecznie dochodząc do końca funkcji `main()`. Wyobraźmy sobie taki przykładowy problem: program ma przeczytać liczbę całkowitą n i dla ujemnej wartości powinien od razu wypisać komunikat NIE, natomiast dla wartości nieujemnej powinien sprawdzić jakąś tam własność liczby n i wypisać komunikat TAK lub NIE. Moglibyśmy skonstruować funkcję `main()` w następujący sposób:

```
int main()
{
  int n;
  cin >> n;
```

```
  if(n < 0)
  {
    cout << "NIE" << endl;
    return 0;
  }
  // dalszy ciąg programu
  . . .
}
```

W ten sposób eliminujemy jeden poziom nawiasów klamrowych w dalszym ciągu programu. Nie jest to może zbyt elegancki sposób zapisu (na pewno nie powinno się go nadużywać), ale może być przydatny.*

A co, gdybyśmy chcieli zakończyć działanie programu w wybranym miejscu, ale nie w funkcji `main()`, tylko w jakiejś innej funkcji? Wtedy należy użyć instrukcji `exit(0)`, która spowoduje przerwanie programu w dowolnym momencie.†

Definicja i deklaracja funkcji

Do tej pory, jeśli pisaliśmy program złożony z kilku funkcji, wtedy zawsze przestrzegaliśmy ich kolejności w tekście programu: zanim funkcja mogła być użyta, najpierw powinna być zdefiniowana. Oto prosty przykład – program obliczający wartość symbolu Newtona dla dwóch liczb naturalnych p, q $(p \geqslant q)$ wpisanych przez użytkownika. Symbol Newtona jest zdefiniowany jako:

$$\binom{p}{q} = \frac{p!}{q! \cdot (p-q)!},$$

natomiast funkcja silnia (**factorial**) jest zdefiniowana jako:

$$\begin{aligned} 0! &= 1, \\ n! &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n \quad (n > 0). \end{aligned}$$

Oto przykładowy program:

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

int factorial(int n)
{
  int result = 1;
  for(int k = 2; k <= n; k++)
    result = result * k;
  return result;
}
```

*Taki trick nazywa się *wyjściem przez komin.*

†W tym przypadku należy dodać jeszcze dyrektywę `#include <cstdlib>` na początku programu.

```
int newton(int p, int q)
{
  return factorial(p) / (factorial(q) * factorial(p - q));
}

int main()
{
  int a, b;
  cin >> a >> b;
  cout << newton(a, b) << endl;
}
```

Najpierw zapisana jest funkcja `factorial()`, potem funkcja `newton()` (bo korzysta z funkcji `factorial()`), a na końcu funkcja `main()` (bo korzysta z funkcji `newton()`).[‡]

Tak być jednak nie musi, gdyż w języku C++ występuje rozróżnienie pomiędzy *deklaracją funkcji* (czyli w uproszczeniu: jej nagłówkiem) oraz *definicją funkcji* (czyli nagłówkiem i całą treścią funkcji).

Deklaracja funkcji `factorial()` wygląda tak:

```
int factorial(int n);
```

lub nawet tak (nazwa argumentu nie jest konieczna, liczy się tylko typ danych):

```
int factorial(int);
```

Natomiast deklaracja funkcji `newton()` wygląda tak:

```
int newton(int p, int q);
```

lub tak:

```
int newton(int, int);
```

Kolejność deklaracji funkcji może być dowolna, ponadto definicje funkcji mogą pojawić się w zasadzie gdziekolwiek (byle po odpowiedniej deklaracji). Zatem nasz program moglibyśmy napisać tak:

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

int factorial(int);
int newton(int, int);
```

[‡]Oczywiście nie należy szaleć z wpisywanymi wartościami, gdyż łatwo doprowadzimy do przepełnienia i niewiarygodnych rezultatów.

```
int main()
{
  int a, b;
  cin >> a >> b;
  cout << newton(a, b) << endl;
}

int factorial(int n)
{
  int result = 1;
  for(int k = 2; k <= n; k++)
    result = result * k;
  return result;
}

int newton(int p, int q)
{
  return factorial(p) / (factorial(q) * factorial(p - q));
}
```

Jeżeli w programie występuje duża ilość funkcji, wtedy opisany powyżej sposób pozwala na uzyskanie bardziej przejrzystego kodu źródłowego.

O ciągach znaków

W języku C++ mamy dwa podstawowe typy danych dla reprezentowania ciągów znaków: typ `string` i tablice znaków (`char []`). Pierwszy typ jest wygodnym w użyciu typem obiektowym (klasą). Łatwo do niego wczytać dane (tekst) i łatwo odwołać się do dowolnego elementu (znaku, typu `char`). Elementy numerowane są jak komórki w tablicy: posiadają indeksy od 0 do `size()-1`.[§] Przy deklarowaniu zmiennej typu `string` nie ma potrzeby podawania jej rozmiaru, gdyż struktura ta dopasowuje się automatycznie do długości tekstu.

Na obiektach typu `string` można między innymi łatwo wykonać operację łączenia (czyli sklejania). Na przykład po wykonaniu instrukcji:

```
string a = "Avada ";
string k = "Kedavra";
cout << a + k << endl;
```

na ekranie pojawi się słynne Zaklęcie Śmierci.

Tablice znaków są swego rodzaju reliktem z języka ANSI C – mają swoje ograniczenia, ale i tak są użyteczne. Przy deklaracji tablicy musimy podać jej rozmiar (nie można go zmienić, jak to w tablicy), na przykład:

```
char text[10];
```

[§]Można także używać równoważnej funkcji `length()`, ale funkcja `size()` jest częściej używana, gdyż jest zdefiniowana dla wszystkich kontenerów z biblioteki STL.

Należy pamiętać, że w takiej zmiennej zmieści się co najwyżej 9-elementowy ciąg znaków, ponieważ ten typ ciągu musi kończyć się znakiem o kodzie ASCII 0 (nie ma tego w typie `string`). Zatem musimy mieć zawsze rezerwę w postaci jednego wolnego miejsca w tablicy. Oczywiście przechowywany ciąg może być krótszy, na przykład po wykonaniu instrukcji:

```
cin >> text;
```

i wpisaniu przez użytkownika ciągu **xyz** zajęte zostaną cztery (!) pierwsze komórki tablicy.[¶]

Długość ciągu znaków w tablicy odczytujemy przy pomocy bibliotecznej funkcji `strlen()` – w powyższym przykładzie na ekranie pojawi się liczba 3 (znak ASCII 0 nie jest wliczany):

```
cout << strlen(text) << endl;
```

Proszę zwrócić uwagę, że typ `char []` nie jest typem obiektowym, więc nie korzystamy z operatora kropki (`.`).[‖]

Na tablicach znaków nie da się bezpośrednio wykonać operacji łączenia przy pomocy znaku `+`, choć oczywiście są do tego odpowiednie funkcje. (Nie będziemy się jednak nimi zajmować.)

Warto wiedzieć, że jeśli w programie pojawia się ciąg znaków w cudzysłowie, na przykład wspomniane już:

```
"Avada "
```

to jest to właśnie tablica znaków (ta zajmuje 7 znaków, pamiętamy o ASCII 0), a nie obiekt typu `string`. Jednak zadbano, aby dla wygody programistów zadziałało przypisanie:

```
string a = "Avada ";
```

Konwersja z tablicy znaków na typ `string` odbywa się automatycznie. W drugę stronę jest odrobinkę trudniej, trzeba posłużyć się funkcją `c_str()`. (Kto ciekawy, niech sobie wygoogla.)

Skoro ciąg znaków zapisany w cudzysłowie nie jest typu `string`, to znaczy, że poniższa instrukcja nie zadziała:

```
cout << "Avada " + "Kedavra" << endl;
```

Dlaczego?

[¶] Należy pamiętać, że znak ASCII 0 to nie to samo, co znak `'0'`, czyli znak cyfry 0.

[‖] W nagłówku programu trzeba dopisać dyrektywę `#include <cstring>`.