Do pełna!

#inicjalizacja #fill
#konstruktor #iota
#resize

Podczas rozwiązywania wielu zadań algorytmicznych pojawia się problem inicjalizacji danych przed przystąpieniem do właściwego rozwiązania. Pamiętamy oczywiście, że sama deklaracja nie wystarczy – z reguły zmienna otrzyma wtedy przypadkową wartość.*

Jeśli chodzi o zmienne *skalarne* (czyli przechowujące pojedynczą, na przykład liczbę całkowitą lub wartość logiczną *true/false*), wtedy sprawa jest prosta: można dokonać tego przez odpowiednie nadanie wartości w deklaracji zmiennej. Dla przykładu:

```
int n = 5;
int k = { };  // nadanie wartości 0
int m = { 1 };
bool p = true;
bool q = { };  // nadanie wartości false
```

Deklaracje zawierające nawiasy klamrowe { } są bardziej trendy.

W przypadku zmiennych strukturalnych sprawa nie jest już taka prosta. Dla przykładu wyzerowanie zadeklarowanej tablicy $A[\,]$ może nastąpić poprzez wykorzystanie odpowiedniej pętli:

```
const int N = 1000;
int A[N];
. . .
for(int i{ }; i < N; i++)
  A[i] = 0;
```

Trudno nazwać taką konstrukcję elegancką, a poza tym mamy wyraźne rozdzielenie: *deklaracja* $\leftrightarrow$ *inicjalizacja*, co wybitnie nie poprawia czytelności kodu.

Jeśli zależy nam na nadaniu wszystkim elementom tablicy wartości zero, wtedy możemy posłużyć się nawiasami klamrowymi:

```
const int N = 1000;
int A[N] = { 0 };
```

*Automatyczna inicjalizacja dotyczy tylko zmiennych deklarowanych globalnie (takich zmiennych raczej unikamy) oraz zmiennych reprezentujących obiekty, na przykład `string` lub `vector<>`.

Zero w nawiasach klamrowych można opuścić:

```
const int N = 1000;
int A[N] = { };
```

Należy wspomnieć, że użycie innej wartości niż 0 w takiej inicjalizacji nie spowoduje bynajmniej wypełnienia całej tablicy podaną liczbą, na przykład deklaracja

```
const int N = 1000;
int B[N] = { 5 };
```

zaowocuje przypisaniem $B[0] = 5$, natomiast wszystkie pozostałe elementy tablicy otrzymają i tak wartość 0 (!).

Inicjalizacja kontenera

Ponieważ najczęściej używaną strukturą danych jest kontener `vector<>`, warto dokładniej przyjrzeć się sposobom jego inicjalizacji. Podczas standardowej deklaracji wektora otrzymujemy kontener o rozmiarze 0, więc trudno coś tu mówić o inicjalizacji:

```
vector<int> V;
```

Jeśli w deklaracji podamy rozmiar wektora, wtedy jego elementy są automatycznie inicjalizowane (`int` na 0, `bool` na *false*, `string` na pusty ciąg znaków itd.):

```
vector<int> P(10);
vector<bool> Q(15);
vector<string> R(20);
```

Rzecz jasna pamiętamy, że rozmiar wektora w jego deklaracji podajemy w nawiasach okrągłych `( )`, a nie w kwadratowych `[ ]` (jak w przypadku tablicy), gdyż w tym przypadku oznacza to wywołanie konstruktora klasy `vector` z odpowiednim argumentem.

W deklaracji wektora możemy podać jeszcze drugi argument kontruktora – będzie to wartość, jaką wektor zostanie wypełniony, na przykład:[†]

```
vector<int> W(30, -1);
```

Powyżej zadeklarowaliśmy trzydziestoelementowy wektor W, którego każdy element jest równy -1.

Jeśli zdarzy nam się zmieniać rozmiar wektora przy pomocy funkcji `resize()`, wtedy – standardowo – podajemy jako argument tej funkcji nowy rozmiar wektora, ale możemy również podać wartość do inicjalizacji. W tym przykładzie wektor *seven* otrzyma rozmiar 3 i zostanie wypełniony wartością 7 (zostanie to wypisane na ekranie):

[†]Podana wartość musi być jako tako zgodna z zadeklarowanym typem elementów wektora.

```
vector<int> seven;
seven.resize(3, 7);
for(int x : seven)
  cout << x << ' ';
```

Inicjalizacja wartości przy użyciu funkcji `resize()` ogranicza się do nowo powstałych elementów wektora. W powyższym przykładzie był to cały wektor, ale tak być nie musi:

```
vector<int> T(5);
T.resize(10, 1);
for(int y : T)
  cout << y << ' ';
```

Na ekranie pojawi się wydruk:

```
0 0 0 0 0 1 1 1 1 1
```

Zera są rezultatem deklaracji w pierwszym wierszu, natomiast funkcja `resize()` jest odpowiedzialna za dopisanie wartości 1 na dalszych pozycjach.

Jeśli chcemy wypełnić nową wartością cały, istniejący już wektor, wtedy lepiej posłużyć się funkcją `fill()`, o której piszemy w jednej z następnych sekcji.

Wypełnianie `string`-u

Typ danych `string` posiada bardzo podobne mechanizmy wypełniania całej struktury, na przykład poniższa deklaracja nadaje zmiennej z wartość ciągu znaków **ggggg**:

```
string z(5, 'g');
```

Konstruktor klasy `string` można zresztą wykorzystać także i bez deklaracji. Poniższa instrukcja spowoduje nadanie zmiennej t wartości dziesięciu plusów:

```
string t;
. . .
t = string(10, '+');
```

Zauważmy, że drugi argument konstruktora to pojedynczy znak (typ `char`), a nie ciąg znaków.

Funkcja `fill()`

Wypełnienie całego wektora (ogólniej: kontenera) można zrealizować przy użyciu uniwersalnej funkcji `fill()`, której podajemy zakres do wypełnienia *od-do* (w formie iteratorów) oraz wartość do wstawienia. Na przykład poniższy pięciolementowy wektor `string`-ów K zostanie wypełniony ciągami **xyz**:

```
vector<string> K(5);
. . .
fill(K.begin(), K.end(), "xyz");
for(string s : K)
  cout << s << ' ';
```

Na ekranie pojawi się wydruk:

```
xyz xyz xyz xyz xyz
```

Najczęściej używa się tej funkcji do wypełniania całego kontenera, ale można – w razie potrzeby – podać bardziej selektywny zakres iteratorów.

Funkcja `fill()` nadaje się także do wypełniania oldschoolowych tablic, wówczas zamiast iteratorów używamy dobrze nam znanych wskaźników, na przykład:

```
const int N = 1000;
int B[N];
. . .
fill(B, B + N, 5);
```

Tablica B zostanie w całości wypełniona wartością 5.

Funkcja `iota()`

Dla koneserów mamy perełkę wywodzącą się jeszcze z kultowego języka APL:[‡] funkcję `iota()`, którą wykorzystuje się do wypełniania kontenera ciągiem wartości. Zazwyczaj chodzi o to, aby wypełnić strukturę kolejnymi liczbami całkowitymi, na przykład 1, 2, 3, ... itd., ale w ogólności funkcja ta oblicza kolejny element przy pomocy operatora inkrementacji `++`, a ten może być zdefiniowany dość dowolnie. (Wystarczy przytoczyć przykład iteratorów – w tym przypadku inkrementacja oznacza przejście na kolejny element kontenera.)

Argumenty funkcji `iota()` są podobne, jak funkcji `fill()`, z tym, że trzecim argumentem jest początkowy wyraz ciągu, którym wypełniany będzie kontener. Dla przykładu, poniższy czteroelementowy wektor L zostanie wypełniony wartościami 1, 2, 3, 4 (wartości te zostaną następnie wypisane na ekranie)):

```
vector<int> L(4);
. . .
iota(L.begin(), L.end(), 1);
for(int m : L)
  cout << m << ' ';
```

Nazwa funkcji pochodzi oczywiście z języka greckiego – to nazwa litery ι (bez kropki – tak też jest oznaczana w języku APL).

[‡]A Programming Language – język ten (powstały w latach 60. ubiegłego wieku – autorem był Kenneth E. Iverson) wyróżniał się absolutnie niezwykłą składnią, której zwięzłości nie był w stanie przebić żaden inny (używany) język programowania.