

Tablice i struktury

Wstęp do Informatyki i Programowania

Maciek Gębala

23 października 2025

Złożone typy danych

Tablica

Przechowuje elementy jednego typu indeksowane typem porządkowym (najczęściej liczbami).

Struktura jednorodna, homogeniczna.

Struktura

Przechowuje elementy różnych typów identyfikowane przez nazwy.

Struktura niejednorodna, heterogeniczna.

Złożone typy danych

Tablica

Przechowuje elementy jednego typu indeksowane typem porządkowym (najczęściej liczbami).

Struktura jednorodna, homogeniczna.

Struktura

Przechowuje elementy różnych typów identyfikowane przez nazwy.

Struktura niejednorodna, heterogeniczna.

Złożone typy danych

Tablica

Przechowuje elementy jednego typu indeksowane typem porządkowym (najczęściej liczbami).

Struktura jednorodna, homogeniczna.

Struktura

Przechowuje elementy różnych typów identyfikowane przez nazwy.

Struktura niejednorodna, heterogeniczna.

Złożone typy danych

Tablica

Przechowuje elementy jednego typu indeksowane typem porządkowym (najczęściej liczbami).

Struktura jednorodna, homogeniczna.

Struktura

Przechowuje elementy różnych typów identyfikowane przez nazwy.

Struktura niejednorodna, heterogeniczna.

Tablice są zawsze indeksowane od 0, operator dostępu to `[]`, rozmiar podawany przy tworzeniu (nie musi być stały), brak kontroli zakresów i informacji o wielkości.

W języku C tablice do funkcji są przekazywane przez nazwę (adres). Nie są więc kopiowane i są zmieniane globalnie.

Struktury to kolekcja danych z nazwami - ułatwiają organizację danych. Mogą być nazwane i traktowane jak pojedynczy typ prosty (typedef).

Tablice są zawsze indeksowane od 0, operator dostępu to `[]`, rozmiar podawany przy tworzeniu (nie musi być stały), brak kontroli zakresów i informacji o wielkości.

W języku C tablice do funkcji są przekazywane przez nazwę (adres). Nie są więc kopiowane i są zmieniane globalnie.

Struktury to kolekcja danych z nazwami - ułatwiają organizację danych. Mogą być nazwane i traktowane jak pojedynczy typ prosty (`typedef`).

Tablice są zawsze indeksowane od 0, operator dostępu to `[]`, rozmiar podawany przy tworzeniu (nie musi być stały), brak kontroli zakresów i informacji o wielkości.

W języku C tablice do funkcji są przekazywane przez nazwę (adres). Nie są więc kopiowane i są zmieniane globalnie.

Struktury to kolekcja danych z nazwami - ułatwiają organizację danych. Mogą być nazwane i traktowane jak pojedynczy typ prosty (`typedef`).

Przykład w języku C

example.c

```
1  #include <stdio.h>
2  #include <math.h>
3  typedef struct point {
4      float x;
5      float y;
6  } point;
7  typedef point segment[2];
8  float lengthp(point a, point b) {
9      return sqrt( (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y) );
10 }
11 float lengths(segment s) {
12     return sqrt( (s[0].x-s[1].x)*(s[0].x-s[1].x)+
13                 (s[0].y-s[1].y)*(s[0].y-s[1].y) );
14 }
15 int main() {
16     point a, b;
17     segment c;
18     a.x = -1.0;  a.y = -2.0;
19     b.x = 3.0;   b.y = 1.0;
20     c[0] = a;    c[1] = b;
21     printf("%f\n%f\n", lengthp(a, b), lengths(c));
22     return 0;
23 }
```

Tablice są indeksowane dowolnie, operator dostępu to `()`, rozmiar podawany przy tworzeniu (nie musi być stały, może zależeć od parametrów wejściowych podprogramu), pełna kontrola zakresów, można pobrać zakres i długość tablicy.

Struktury są tworzone słowem `record`.

Szablony struktur i tablic mogą być nazwane i traktowane jak pojedynczy typ prosty (`type`).

Tablice są indeksowane dowolnie, operator dostępu to `()`, rozmiar podawany przy tworzeniu (nie musi być stały, może zależeć od parametrów wejściowych podprogramu), pełna kontrola zakresów, można pobrać zakres i długość tablicy.

Struktury są tworzone słowem `record`.

Szablony struktur i tablic mogą być nazwane i traktowane jak pojedynczy typ prosty (type).

Tablice są indeksowane dowolnie, operator dostępu to `()`, rozmiar podawany przy tworzeniu (nie musi być stały, może zależeć od parametrów wejściowych podprogramu), pełna kontrola zakresów, można pobrać zakres i długość tablicy.

Struktury są tworzone słowem `record`.

Szablony struktur i tablic mogą być nazwane i traktowane jak pojedynczy typ prosty (`type`).

Przykład w języku Ada

example.adb

```
1 with Ada.Text_IO;
2 use Ada.Text_IO;
3 with Ada.Numerics.Elementary_Functions;
4 use Ada.Numerics.Elementary_Functions;
5
6 procedure example is
7     type Point is record
8         x : Float := 0.0;
9         y : Float := 0.0;
10    end record;
11
12    type Segment is array (1 .. 2) of Point;
13
14    function length (s : Segment) return Float is
15    begin
16        return Sqrt ((s (1).x - s (2).x)**2 +
17                      (s (1).y - s (2).y)**2);
18    end length;
19
20    function length (a : Point; b : Point) return Float is
21    begin
22        return Sqrt ((a.x - b.x)**2 + (a.y - b.y)**2);
23    end length;
```

Przykład w języku Ada

example.adb

```
25   a, b : Point;  
26   c : Segment;  
27 begin  
28   a.x := -1.0;  
29   a.y := -2.0;  
30   b.x := 3.0;  
31   b.y := 1.0;  
32   c (1) := a;  
33   c (2) := b;  
34   Put_Line (Float'Image (length (a, b)));  
35   Put_Line (Float'Image (length (c)));  
36 end example;
```

Język Python nie ma tablic ani struktur. Zastępują je heterogeniczne listy i krotki (ale bez nazw pól).

Listy inicjujemy przez podstawienie `[]` (listy pustej) lub ciągu elementów w tych nawiasach. Elementy dodajemy metodą `append` - lista ma dynamiczną długość. Elementy są indeksowane od zera i możemy się do nich odwołać operatorem dostępu `[]`. Funkcja `len` na liście zwraca jej długość.

Język Python nie ma tablic ani struktur. Zastępują je heterogeniczne listy i krotki (ale bez nazw pól).

Listy inicjujemy przez podstawienie `[]` (listy pustej) lub ciągu elementów w tych nawiasach. Elementy dodajemy metodą `append` - lista ma dynamiczną długość. Elementy są indeksowane od zera i możemy się do nich odwołać operatorem dostępu `[]`. Funkcja `len` na liście zwraca jej długość.

Przykład użycia tablic: Sito Eratostenesa

Prosta metoda znajdowania liczb pierwszych na przedziale od 2 do ustalonego n .

Pseudokod algorytmu

```
1: for  $i \leftarrow 2 \dots n$  do
2:    $sito[i] \leftarrow true$ 
3: end for
4: for  $i \leftarrow 2 \dots n$  do
5:   if  $sito[i]$  then
6:      $j \leftarrow i + i$ 
7:     while  $j \leq n$  do
8:        $sito[j] \leftarrow false$ 
9:        $j \leftarrow j + i$ 
10:    end while
11:  end if
12: end for
```

Po zakończeniu algorytmu p jest pierwsze jeśli $sito[p] = true$.

Przykład użycia tablic: Sito Eratostenesa

Prosta metoda znajdowania liczb pierwszych na przedziale od 2 do ustalonego n .

Pseudokod algorytmu

```
1: for  $i \leftarrow 2 \dots n$  do
2:    $sito[i] \leftarrow true$ 
3: end for
4: for  $i \leftarrow 2 \dots n$  do
5:   if  $sito[i]$  then
6:      $j \leftarrow i + i$ 
7:     while  $j \leq n$  do
8:        $sito[j] \leftarrow false$ 
9:        $j \leftarrow j + i$ 
10:    end while
11:  end if
12: end for
```

Po zakończeniu algorytmu p jest pierwsze jeśli $sito[p] = true$.

Przykład użycia tablic: Sito Eratostenesa

Prosta metoda znajdowania liczb pierwszych na przedziale od 2 do ustalonego n .

Pseudokod algorytmu

```
1: for  $i \leftarrow 2 \dots n$  do  
2:    $sito[i] \leftarrow true$   
3: end for  
4: for  $i \leftarrow 2 \dots n$  do  
5:   if  $sito[i]$  then  
6:      $j \leftarrow i + i$   
7:     while  $j \leq n$  do  
8:        $sito[j] \leftarrow false$   
9:        $j \leftarrow j + i$   
10:    end while  
11:  end if  
12: end for
```

Po zakończeniu algorytmu p jest pierwsze jeśli $sito[p] = true$.

Problem do implementacji

Przy pomocy sita Eratostenesa napisać funkcję która dla podanego n zwróci liczbę liczb pierwszych na przedziale od 2 do n .

Implementacja w C

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  void compute_sieve(bool s[], unsigned n) {
5      unsigned i, j;
6      for ( i=2; i <= n; i++ ) s[i] = true;
7      for ( i=2; i <= n; i++ )
8          if ( s[i] )
9              for ( j=i+i; j <= n; j+=i )
10                 s[j] = false;
11 }
12 unsigned count_primes(bool s[], unsigned n) {
13     int i, c = 0;
14     for ( i=2; i <= n; i++ )
15         if ( s[i] ) c++;
16     return c;
17 }
18 unsigned primenumbers(unsigned n) {
19     bool sieve[n+1];
20     compute_sieve(sieve, n);
21     return count_primes(sieve, n);
22 }
```

Implementacja w C

```
23 int main() {  
24     unsigned n;  
25  
26     scanf("%u", &n);  
27     printf("%u\n", primenumbers(n));  
28  
29     return 0;  
30 }
```

Implementacja w Adzie

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3 procedure PrimeNumbers is
4     function PrimeNumbers (n : Natural) return Natural is
5         type Sieve is array (2 .. n) of Boolean;
6
7         procedure ComputeSieve (s : in out Sieve) is
8             j : Natural;
9         begin
10             for i in s'Range loop
11                 s (i) := True;
12             end loop;
13             for i in s'First .. s'Last loop
14                 if s (i) then
15                     j := i + i;
16                     while j <= n loop
17                         s (j) := False;
18                         j := j + i;
19                     end loop;
20                 end if;
21             end loop;
22         end ComputeSieve;
```

Implementacja w Adzie

```
24     function CountPrimes (s : Sieve) return Natural is
25         c : Natural := 0;
26     begin
27         for i in s'First .. s'Last loop
28             if s (i) then
29                 c := c + 1;
30             end if;
31         end loop;
32         return c;
33     end CountPrimes;
34
35     s : Sieve;
36 begin
37     ComputeSieve (s);
38     return CountPrimes (s);
39 end PrimeNumbers;
40
41     n : Natural;
42 begin
43     Get (n);
44     Put_Line (Integer'Image (PrimeNumbers (n)));
45 end PrimeNumbers;
```

Implementacja w Pythonie

```
1 def primenumbers(n) :
2     def create_sieve(s, n) :
3         for _ in range(n+1) :
4             s.append(True)
5         s[0] = False
6         s[1] = False
7     def compute_sieve(s) :
8         for i in range(2, len(s)) :
9             if s[i] :
10                 j = i + i
11                 while j < len(s) :
12                     s[j] = False
13                     j = j + i
14     def count_primes(s) :
15         c = 0
16         for e in s :
17             if e :
18                 c = c + 1
19         return c
20
21     s = []
22     create_sieve(s, n)
23     compute_sieve(s)
24     return count_primes(s)
```

Implementacja w Pythonie

```
26 def main() :  
27     n = int(input(""))  
28     print( primenumbers(n) )  
29  
30 if __name__ == "__main__":  
31     main()
```