10. Enodimenzionalna polja

1

10.1. Splošno o poljih

2

- do sedaj:
  - vsaka spremenljivka je predstavljala en podatek
- **polje** je zbirka podatkov istega tipa
  - vsak podatek ima natančno določeno mesto oz. indeks
  - prvi element je na indeksu 0 (nič)
  - do elementov dostopamo preko indeksov
- polje omogoča **direkten** dostop do podatkov

Še o poljih

3

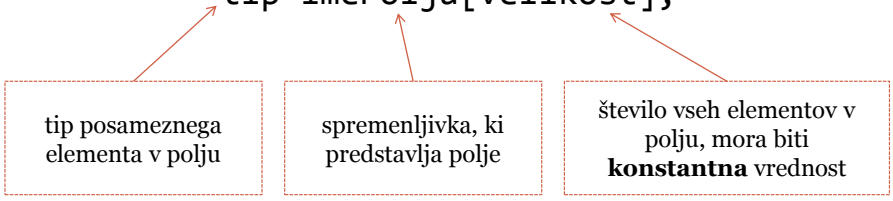
- C++ omogoča uporabo polj poljubnega tipa
  - torej lahko imamo polje polj
- ločili bomo dva načina uporabe polj
  - odvisno od tega, kako se rezervira pomnilnik za polje
- govorili bomo o
  - statičnih poljih
  - dinamičnih poljih

11.2. Statična polja

4

- značilno za statična polja:
  - ob sami deklaraciji polja **moramo** podati število elementov
  - velikost takega polja ostane skozi ves doseg nespremenjena
  - torej velikost polja je znana že, ko programiramo
- statično polje elementov tipa `tip` deklariramo tako:

```
tip imePolja[velikost];
```



Dostop do elementov v polju

5

- do posameznega elementa v polju dostopamo tako:
  - ob imenu polja v oglatih oklepajih podamo indeks elementa

imePolja[indeks]

- indeks je poljuben celoštevilski izraz
- sami moramo paziti da
  - element s podanim indeksom obstaja
    - najmanjši indeks je `0`
    - največji indeks je `(velikostPolja - 1)`

Dostop do elementov v polju (2)

6

- do posameznega elementa v polju dostopamo tako:

`imePolja[indeks]`

PRIMER:

```
int polje [5];
polje [0] = 10;
polje [polje[0]-8]=5;
```

deklariramo polje `5` elementov, indeksirani so od `0` do `4`, posamezni element je celo število.

v element polja z indeksom `0` shranimo vrednost `10`

v element polja z indeksom `2` shranimo `5`

Primer 01

7

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

const int VELIKOST = 5;

int main()
{
    int polje[VELIKOST];

    for(int i=0; i<VELIKOST; i++) {
        polje[i] = i;
        cout << "polje[" << i << "]=";
        cout << polje[i] << endl;
    }

    return 0;
}
```

definiramo celoštevilsko konstanto – njene vrednosti ne moremo spremeniti

deklariramo polje 5 celih števil, elementi nimajo določene vrednosti

i teče po vseh indeksih polja

v polje na ustrezno mesto shranimo vrednost indeksa

Inicializacija elementov v polju

8

- elemente polja lahko ob deklaraciji tudi inicializiramo na naslednji način:
  - int polje[3] = {5, 100, -7};
  - int polje[20] = {1, 3};
  - int polje[] = {6, 3, 2};

za vse elemente polja podamo vrednosti

podamo vrednosti prvih nekaj elementov, ostali so privzete vrednosti za ustrezen podatkovni tip

velikost polja se določi na podlagi števila podanih elementov, v tem primeru 3 elementi

Polja kot parametri funkcij

- polja so lahko tudi parametri funkcij
- imamo dve možnosti:
  1. v parametru za polje podamo celotno deklaracijo polja
  2. v parametru za polje nakažemo, da gre za polje, in dodamo dodaten parameter za število elementov v polju
     - C++ dopušča, da pri poljih kot parametrih funkcije, izpustimo prvo (če jih je več, ostale moramo podati) dimenzijo, v tem primeru podatke o tej dimenziji podamo preko dodatnega parametra
- glej naslednja dva primera

Primer 02


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

const int VEL = 10;

// Funkcija prebere vrednosti elt. polja
void preberi(int aP[VEL]) {
    for(int i=0; i<VEL; i++) {
        cout << "Vnesi element [" << i << "]: ";
        cin >> aP[i];
    }
}

int main()
{
    int x[VEL], y[VEL]; // imamo dve polji velikosti VEL
    preberi(x);
    preberi(y);

    return 0;
}
```

slabost tega načina je, da bo funkcija pravilno delovala le za polja iste dimenzije

Primer 03


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

const int VEL = 10;

// Funkcija prebere vrednosti elt. polja
void preberi(int aP[], int aN) {
    for(int i=0; i<aN; i++) {
        cout << "Vnesi element [" << i << "]: ";
        cin >> aP[i];
    }
}

int main()
{
    int x[VEL], y[VEL]; // imamo dve polji velikosti VEL
    preberi(x, VEL);    // kam beremo podatke in koliko jih preberemo
    preberi(y, VEL);

    return 0;
}
```

funkcijo lahko uporabimo za poljubno polje celih števil, z drugim parametrom povemo, koliko števil je v polju

Polja kot parametri funkcij

Opazimo:

- nismo uporabili prenosa parametra (polja) po referenci
  - kot parameter funkcije se ne prenaša celotno polje, temveč le njegov naslov

Dodatna razlaga:

- nismo uporabili prenosa po referenci, zato ker je kazalec aP prvi objekt tega polja.

Slabost statičnih polj

13

- statična polja imajo naslednjo slabost:
  - velikost polja moramo poznati že ob deklaraciji polja
  - tega ne poznamo vedno, zato se lahko zgodi:
    - deklariramo preveliko polje in ostane na pol prazno
      - slaba izkoriščenost pomnilnika
    - deklariramo premajhno polje
      - presežemo rezerviran pomnilnik in pride do napake
- to težavo lahko rešimo z **dinamičnimi polji**

11.3. Kazalčni podatkovni tipi

14

- deklaracija spremenljivke
  - podamo podatkovni tip in identifikator
  - prevajalnik na nekem **naslovu** v pomnilniku rezervira ustrezno količino pomnilnika, kamor se shrani vrednost, ki jo spremenljivka predstavlja
  - količina rezerviranega pomnilnika je odvisna od podatkovnega tipa
  - do vrednosti spremenljivke dostopamo z imenom spremenljivke

Še o kazalčnih tipih (1)

15

- dostopamo lahko tudi do naslova, kjer je vrednost spremenljivke shranjena
- naslovu pomnilniške lokacije rečemo tudi **referenca**
- v C++ referenco dobimo z operatorjem &

```
&ime
```

vrne naslov lokacije v pomnilniku, kjer je shranjena vrednost spremenljivke ime

Še o kazalčnih tipih (2)

16

- **kazalec** imenujemo spremenljivko, ki hrani naslov pomnilniške lokacije, kjer je podatek
- kazalec deklariramo na naslednji način

```
tip *ime;
```

PRIMER:

```
int *kazalec;
```

kazalec je spremenljivka, ki hrani naslov neke pomnilniške lokacije, kjer je shranjen podatek tipa `int`

Kazalci in povezani operatorji (1)

17

PRIMER:

```
int stevilo = 10;
int *k;
k = &stevilo;
```

celoštevilska spremenljivka

kazalec na celoštevilsko spremenljivko, ob deklaraciji še ne vemo kam kaže

ali drugače povedano

spremenljivka k hrani naslov pomnilniške lokacije, kjer je shranjeno celo število; ne vemo še, kateri naslov je shranjen

v spremenljivko k shranimo naslov lokacije, kjer je shranjena vrednost spremenljivke `stevilo`

Kazalci in povezani operatorji (2)

18

- kazalec lahko dobi posebno vrednost 0
  - pomeni, da kazalec ne kaže nikamor oz. ni shranjen noben naslov
  - uporabljamo za stražarja
- operator reference & ima svoj nasprotni operator *
  - operator * izvede dereferenco
  - pridemo do vrednosti, na katero kaže kazalec
- sintaksa uporabe operatorja * je naslednja:

```
*kazalec
```

vrednost podatka na naslovu, kamor kaže kazalčna spremenljivka kazalec

Kazalci in povezani operatorji (3)

19

PRIMER:

```
int stevilo = 10;
int *k;
k = &stevilo;
int n = *k + 1;
```

spremenljivka n dobi vrednost 11

- kazalci v C++ so tesno povezani s polji
- polje v C++ lahko imamo za posebno vrsto kazalca
  - kazalec, ki kaže na začetek neke količine pomnilnika, kamor shranjujemo indeksirane elemente

11.4. Dinamična polja

20

- pogosto ne vemo v naprej, kako veliko polje bomo potrebovali v programu
  - lahko pa to informacijo dobimo med izvajanjem programa.
- izkoristili bomo možnost, ki jo ponuja C++, da programer sam rezervira pomnilnik za polje podatkov
- za to potrebujemo kazalce:
  - v kazalcu bomo shranili naslov prvega elementa polja
  - vsi ostali elementi si v pomnilniku sledijo zaporedoma

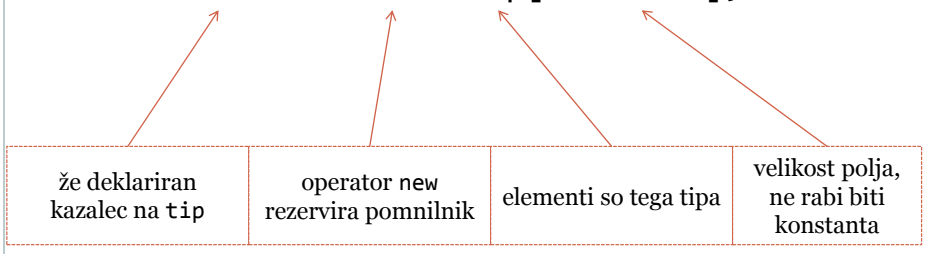
Dinamična polja – rezervacija pomnilnika

21

- za deklaracijo dinamičnih polj, bomo uporabili operator `new[]`

Sintaksa rezervacije pomnilnika za polje:

```
kazalec = new tip[velikost];
```



Dinamična polja – primer

22

PRIMER:

```
int *polje;
int velikost;
cout << "Koliko elt. bo v polju? ";
cin >> velikost;

polje = new int[velikost];
polje[0] = 3;   // do elementa dinamičnega polja
                // dostopamo enako kot do statičnega
```

Dinamična polja – sproščanje pomnilnika

23

- dinamična polja zasedajo del pomnilnika, ki ga prevajalnik nameni za dinamične podatkovne strukture
- ko dinamičnega polja ne potrebujemo več, je smiselno nezasedeni prostor sprostiti
- to naredimo z operatorjem `delete[]`

SINTAKSA:

```
delete[] kazalec;
```

Operatorja new in delete

24

- operatorja new in delete lahko uporabljamo tudi za to, da rezerviramo oz. sprostimo **en sam** element nekega tipa
- v tem primeru uporabimo naslednjo sintakso:

```
kazalec = new tip;
delete kazalec;
```

- z oglatimi oklepaji pri teh dveh operatorjih torej nakažemo, da gre za več kot en podatek

Primer 03


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

int main() {
    // Preberemo velikost dinamičnega polja
   int velikost;
   cout << "Koliko elementov zelis imeti v polju? ";
   cin >> velikost;

    // deklariramo kazalec na prvi element polja celih števil
    // in rezerviramo pomnilnik za "velikost" elementov
   int *polje = new int [velikost];

   for (int i = 0; i<velikost; i++) {
          polje [i] = i * i;
          cout << "Polje [" << i << "]=" << polje [i] << endl;
   }

    // sprostimo rezervirani pomnilnik
   delete[] polje;

   return 0;
}
```

Dinamična polja kot parametri funkcij

- dinamično polje kot parameter funkcije
  - dejansko ne podajamo za parameter polja
  - podajamo le kazalec na prvi element polja
  - torej vse spremembe nad poljem se izvedejo nad isto lokacijo v pomnilniku

Primer 04


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

void preberi(int *aP, int aN) {
    for(int i=0; i<aN; i++) {
        cout << "polje[" << i << "]= ";
        cin >> aP[i];
    }
}

void izpis(int *aP, int aN) {
    for(int i=0; i<aN; i++) {
        cout << aP[i] << " ";
    }
    cout << endl;
}
```

```
int main()
{
    int velikost;
    cout << "Koliko elt. bo v polju? ";
    cin >> velikost;

    int *mojePolje = new int[velikost];
    preberi(mojePolje, velikost);
    izpis(mojePolje, velikost);
    delete[] mojePolje;
    return 0;
}
```