

RI301

Strukture podataka

dr.sc. Edin Pjanić

Pregled predavanja

- Motivacija / izazov
- Apstraktni tip podatka: nešto kao niz, slično kao `std::array`
- Implementacija

Motivacija ili izazov

Pretpostavimo da imamo sljedeću vrstu problema:

- Vodimo evidenciju o studentima (brojevi indeksa u formatu FET-a).
- Želimo na jednostavan i brz način pristupiti svakom studentu po broju indeksa.

Mogli bismo ovo riješiti čuvanjem studenata (objekata) u C++ nizu i pristupanjem po indeksu studenta tako da indeks studenta odgovara indeksu elementa u nizu. Imali bismo brz pristup.

Međutim, da li je ovo najsretnije rješenje?

Pretpostavimo da imamo drugu vrstu problema:

- U nekom fajlu su mjerene temperature u podne za svaki dan u određenom periodu, zaokružena na cijeli broj.
- U fajlu su u prvom redu dati najmanja i najveća mjerena temperatura pa su onda u ostalim redovima upisani datum i temperatura mjerena tog dana.
- Potrebno je odrediti ukupan broj dana u kojima je dnevna temperatura bila svaka od mjerenih vrijednosti.

Kako bismo ovo mogli riješiti?

Nizovi


- Standardni C++ niz (array):
 - implementiran preko sukcesivnih memorijskih lokacija u kojima su elementi istog tipa
 - pristup elementima preko indeksa
 - za n elemenata, indeksi idu od 0 do $n - 1$
 - pri pristupu elementima ne provjerava se valjanost indeksa
 - kopiranje nizova nije jednostavno (npr. `a=b;`)
 - nije moguće mijenjati veličinu niza
- Želimo “**naš niz**” kao ATP – sa zahtjevima koji nam trebaju:
 - pristup elementima preko indeksa uz provjeru ispravnosti indeksa prilikom pristupa elementima
 - indeksi elemenata u proizvoljnom opsegu (npr. od -40 do 50)
 - mogućnost promjene veličine vektora i opsega indeksa prema potrebi
 - mogućnost ubacivanja novog elementa na kraj vektora i automatsko povećanje veličine vektora za 1 (`push_back`)
 - mogućnost jednostavnog kopiranja niza pomoću operatora dodjeljivanja
- Nazovimo ovakvu strukturu (radni naziv): Naš niz

Naš Niz (ATP)

- ATP niz:

Elementi : $\langle a_d, a_{d+1}, a_{d+2}, a_{d+3}, \dots, a_{g-1}, a_g \rangle$

Indeksi: $d \quad d+1 \quad d+2 \quad \quad \quad g-1 \quad g$

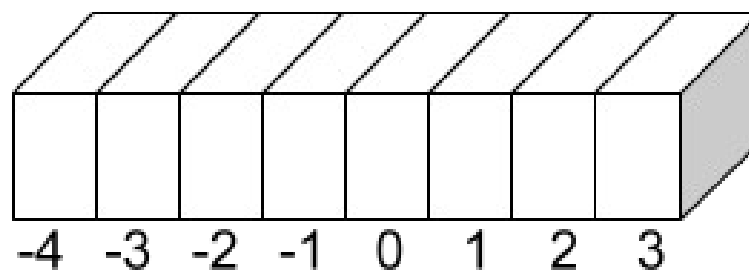
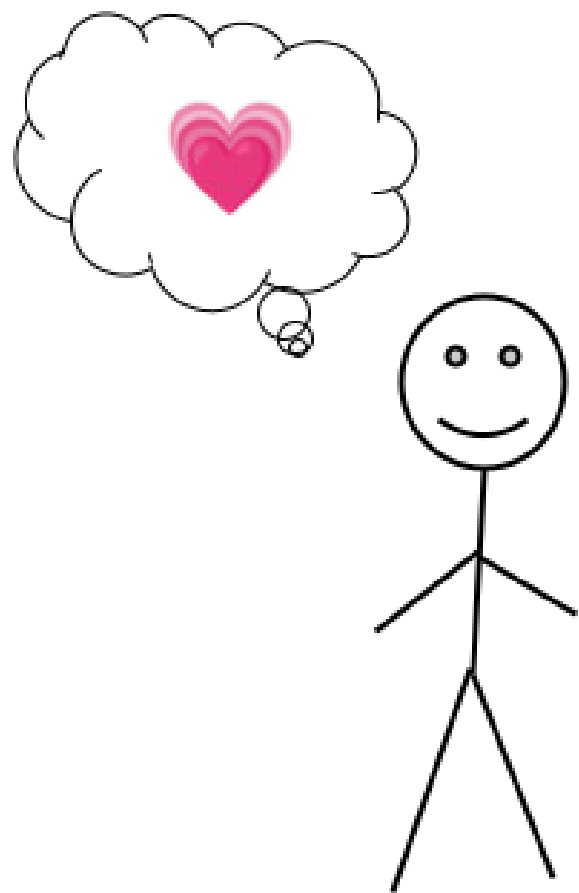


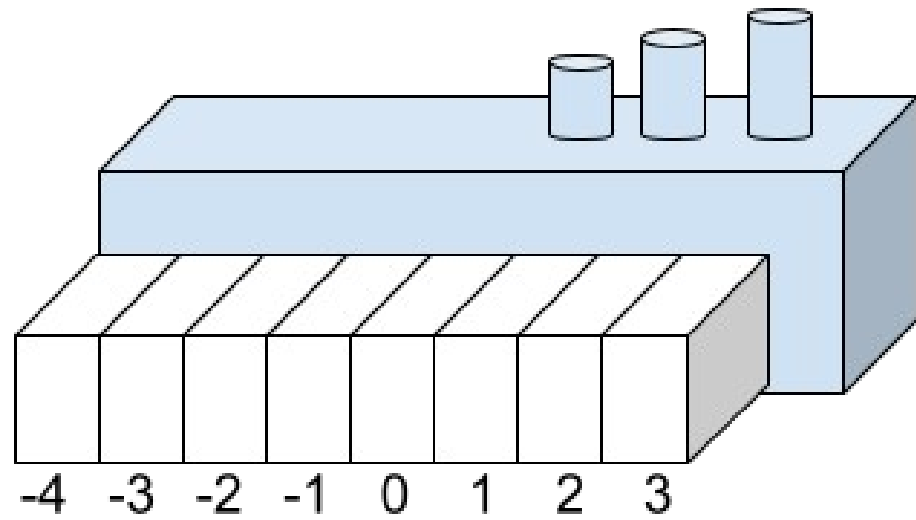
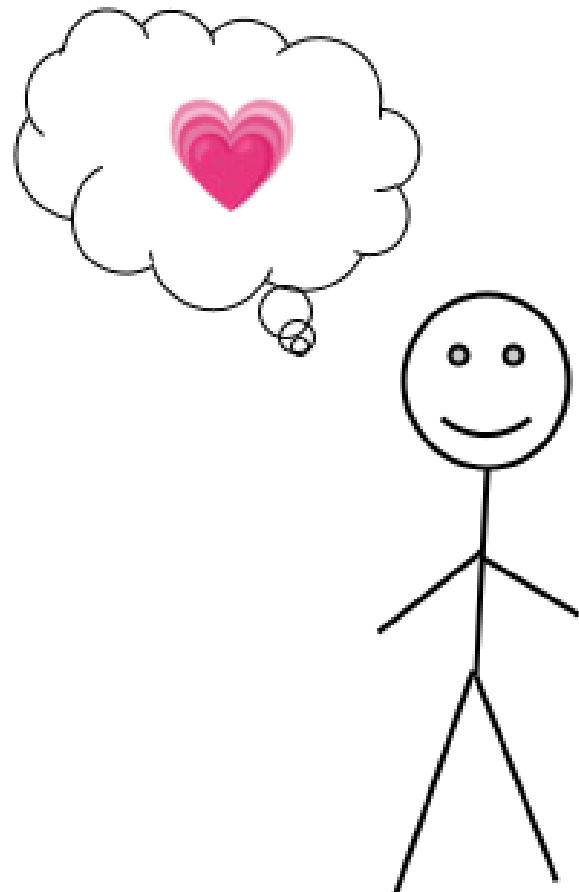
gdje su:

d – donja granica indeksa (najmanji indeks)

g – gornja granica indeksa (najveći indeks)

- Kako implementirati ovakvu strukturu?
 - Možemo dinamički alocirati potrebnu količinu memorije i tu memoriju posmatrati kao standardni C++ niz.



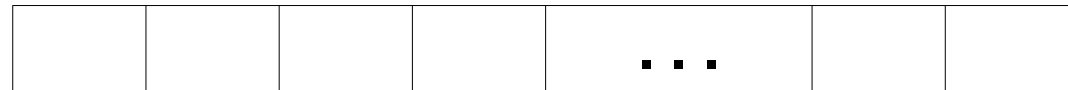


Naš Niz - implementacija

- ATP niz:

Elementi : $\langle a_d, a_{d+1}, a_{d+2}, a_{d+3}, \dots, a_{g-1}, a_g \rangle$

Indeksi: $d \quad d+1 \quad d+2 \quad \dots \quad g-1 \quad g$



Indeksi C++ niza: $0 \quad 1 \quad 2 \quad 3 \quad \dots \quad n-2 \quad n-1$

detalji
implementacije

Naš Niz – primjer niza cijelih brojeva

- ATP niz:

Indeksi ATP niza:

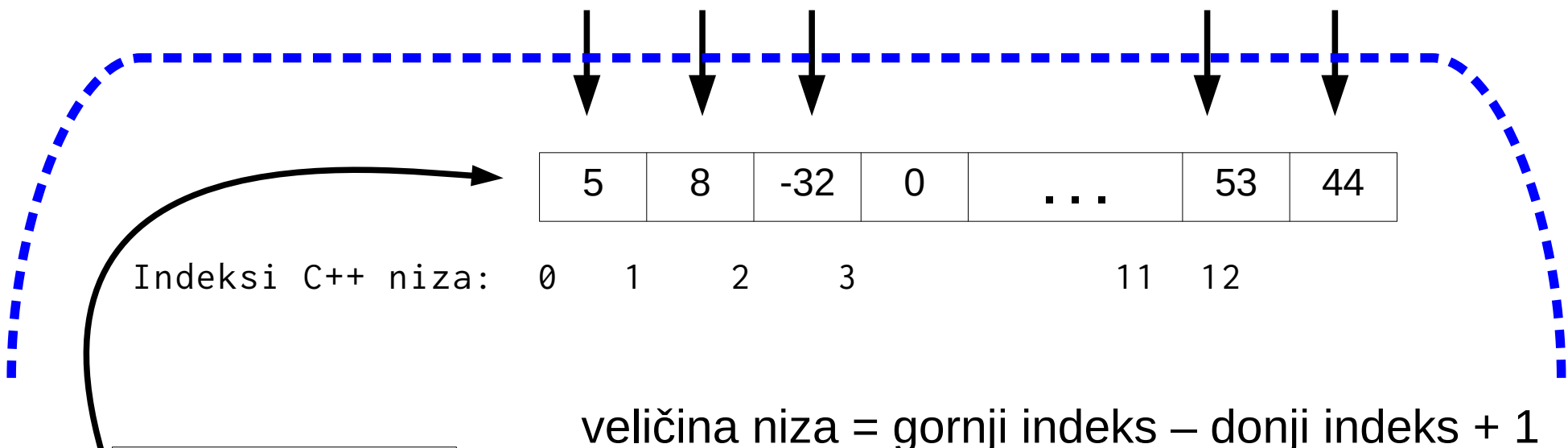
donji indeks

gornji indeks

10 11 12

21 22

Elementi : < 5, 8, -32, 0, ..., 53, 44 >



5	8	-32	0	...	53	44
---	---	-----	---	-----	----	----

Indeksi C++ niza:

0 1 2 3

11 12

dinamički
alociran niz
elemenata

veličina niza = gornji indeks – donji indeks + 1

Kako pristupati elementima ove strukture korištenjem indeksa?

- Definisati operator []

Ne miješati pojam indeksa “našeg niza” sa indeksom elemenata u dinamički alociranom nizu.

Naš Niz - implementacija

Implementiraćemo Naš Niz u klasi koju ćemo, radi jednostavnosti, nazvati **Niz**.

```
template <typename Elem>  
class Niz  
{
```

Naš Niz – implementacija nekih metoda

```
template <typename Elem>
Niz<Elem>::Niz(int d, int g)
{
    if(g < d)
        throw std::invalid_argument("Donja granica ne smije biti veca od gornje.");
    velicina = g - d + 1;
    elementi = new Elem[velicina];
    donja = d;
    gornja = g;
}
```

```
template <typename Elem>
Niz<Elem>::~~Niz()
{
    delete [] elementi;
}
```

Pravilo Petorke: Ako je u klasi definisano jedno od dole navedenih, vjerovatno bi trebalo definisati svih pet:

- Destruktor,
- Copy konstruktor,
- Copy operator dodjeljivanja,
- Move konstruktor,
- Move oper. dodjeljivanja.

Naš Niz – implementacija nekih metoda

```
template <typename Elem>
Elem & Niz<Elem>::operator[](int index)
{
    if(index < donja || gornja < index)
    {
        std::string poruka = "Indeks (" + std::to_string(index)
            + ") izvan granica ["
            + std::to_string(donja) + ", " + std::to_string(gornja) + "].";
        throw std::invalid_argument(poruka);
    }
    return elementi[ index - donja ];
}
```

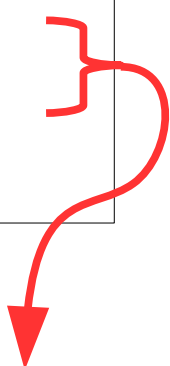
Konstruktor od liste za inicijalizaciju

```
Niz(std::initializer_list<Elem>);
```

- Konstruktor uzima listu za inicijalizaciju i konstruiše naš kontejner na osnovu te liste.
- S obzirom na to da nisu date granice indeksa, početni indeks elemenata niza će biti 0.
- Primjer korištenja:

```
Niz<int> a{5,6,7,4,3,3,44,-44};
```

```
template <typename Elem>
Niz<Elem>::Niz(std::initializer_list<Elem> il)
: velicina(il.size()),
  donja(0), gornja(velicina - 1),
  elementi(new Elem[velicina])
{
    for(auto it=il.begin(), int i=0; it!=il.end(); ++it, ++i)
        elementi[i] = *it;
}
```



```
std::copy(il.begin(), il.end(), elementi);
```

Konstruktor od liste za inicijalizaciju 2

```
Niz(std::initializer_list<Elem>, int);
```

- Konstruktor ovaj put uzima i početni indeks.
- Primjer korištenja:

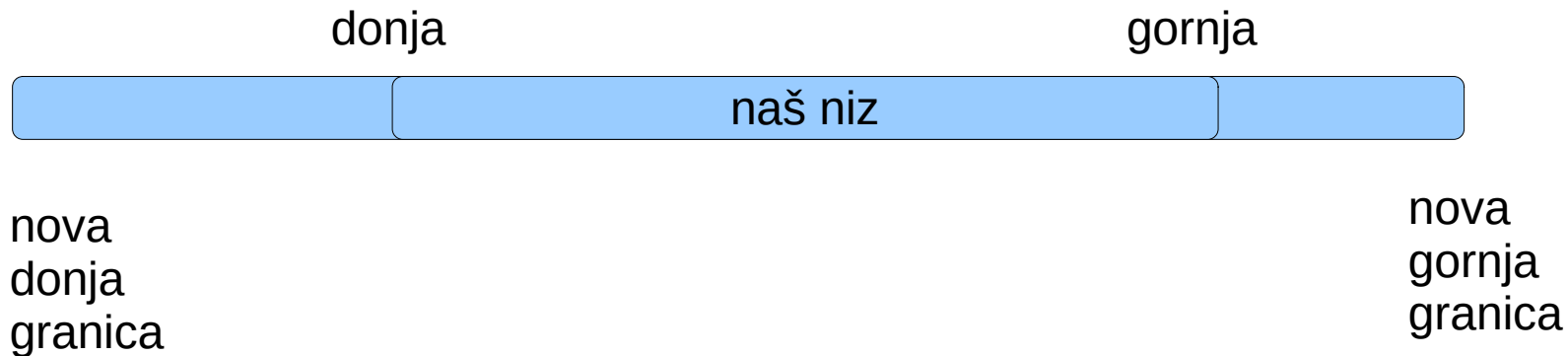
```
Niz<int> a({5,6,7,4,3,3,44,-44}, 10);
```

```
template <typename Elem>
Niz<Elem>::Niz(std::initializer_list<Elem> il, int poc = 0)
: velicina(il.size()),
  donja(poc), gornja(poc + velicina - 1),
  elementi(new Elem[velicina])
{
    std::copy(il.begin(), il.end(), elementi);
}
```

Naš Niz – realociranje

Metod pozivamo kad želimo izmijeniti granice našeg kontejnera (u indeksima):

- Moguće je proširiti ili smanjiti granice (krajnje indekse).
- Pri tome elementi zadržavaju svoje indekse. To znači da se potencijalno neki elementi mogu odbaciti.



```
// Primjer:
```

```
niz.realociraj(nova_donja, nova_gornja);
```

Naš Niz – realociranje

```
template <typename Elem>
void Niz<Elem>::realociraj(int d, int g)
{
    if(g < d)
        throw invalid_argument("Donja granica ne smije biti veca od gornje.");
    Elem * novi_niz = new Elem[g - d + 1];
```

```
    donja = d;
    gornja = g;
}
```

Metod pozivamo kad želimo izmijeniti granice našeg kontejnera (u indeksima):

- Moguće je proširiti ili smanjiti granice (krajnje indekse).
- Pri tome elementi zadržavaju svoje indekse. To znači da se potencijalno neki elementi mogu odbaciti.

Naš Niz – realociranje

```
template <typename Elem>
void Niz<Elem>::realociraj(int d, int g)
{
    if(g < d)
        throw invalid_argument("Donja granica ne smije biti veca od gornje.");
    Elem * novi_niz = new Elem[g - d + 1];

    if(d <= gornja && g >= donja) // Zadržavamo elemente kod kojih su indeksi u novom opsegu. To
    {                               // omogućava prosirenje na proizvoljnu stranu ali i skracenje.
        int d_prenos, g_prenos;
        if(donja < d)
            d_prenos = d;
        else
            d_prenos = donja;

        if(gornja > g)
            g_prenos = g;
        else
            g_prenos = gornja;

        for(int i=d_prenos; i<=g_prenos; i++)
            novi_niz[i-d] = elementi[i-donja];
    }
    delete[] elementi;
    elementi = novi_niz;
    donja = d;
    gornja = g;
}
```

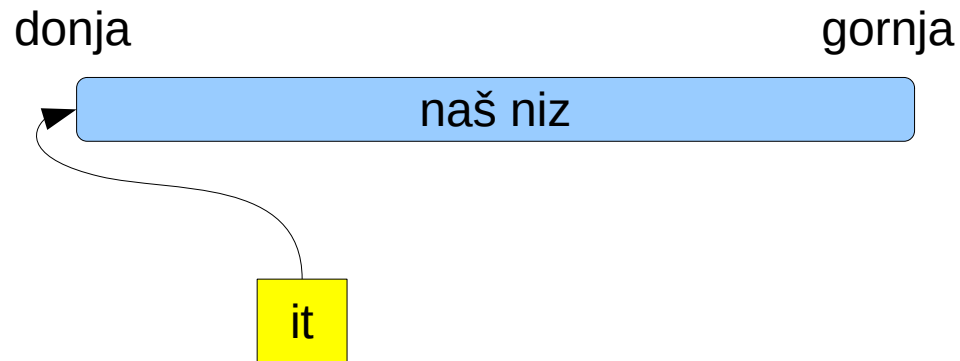
Metod pozivamo kad želimo izmijeniti granice našeg kontejnera (u indeksima):

- Moguće je proširiti ili smanjiti granice (krajnje indekse).
- Pri tome elementi zadržavaju svoje indekse. To znači da se potencijalno neki elementi mogu odbaciti.

Iterator za Naš Niz

Želimo da možemo napisati kod sličan kodu:

```
for(auto it=niz.begin(); it!=niz.end(); it++){  
    ... *it ...  
}
```



```
auto it=niz.begin();
```

```
it++;
```

```
*it = val;
```

```
it--;
```

```
++it;
```

```
--it;
```

```
auto it2 = it;
```

Kako bi bilo najzgodnije da imenujemo tip iteratora?
(analogija sa standardnom bibliotekom)

```
Niz<int>::iterator it = niz.begin();
```

Kako da objekat iteratora:

- dođe do elementa na kojeg “pokazuje”?
- pomjeri se na naredni element?
- pomjeri se na prethodni element?

Iterator za Naš Niz

Želimo da možemo napisati kod sličan kodu:

```
for(auto it=niz.begin(); it!=niz.end(); it++){  
    ... *it ...  
}
```

```
template<typename Elem>  
class Niz<Elem>::iterator  
{  
    private:  
        Elem * pel;  
  
    public:  
        iterator(Elem *p) : pel(p){}  
  
        Elem& operator*(){return *pel; }  
  
        iterator& operator++() {++pel; return *this; }  
        iterator operator++(int) {iterator temp = *this; ++pel; return temp; }  
  
        iterator& operator--() {--pel; return *this; }  
        iterator operator--(int) {iterator temp = *this; --pel; return temp; }  
  
        bool operator!=(const iterator& b) const {return pel != b.pel; }  
        bool operator==(const iterator& b) const {return pel == b.pel; }  
};
```

Naš Niz

```
template<typename T>
class Niz
{
    private:
        int velicina;
        int donja, gornja;
        T * elementi;
        void alociraj(int, int);
        void kopiraj(const Niz&);
    public:
        Niz(int d, int g);
        Niz(int n);
        Niz(const Niz&);
        Niz(Niz&&);
        Niz(std::initializer_list<T>, int=0);
        ~Niz();
        Niz & operator=(const Niz&);
        Niz & operator=(Niz&&);
        T & operator[](int index);
        const T & operator[](int index) const ;
        void realociraj(int d, int g);
        int donja_granica() const {return donja;};
        int gornja_granica() const {return gornja;};
        int velicina_niza() const {return velicina;};
        int size() const {return velicina;};
        class iterator;
        iterator begin(){return iterator(elementi);}
        iterator end(){return iterator(elementi + velicina);}
};
```



Osobine iteratora (pogledati `std::iterator_traits`)

- Da bi naš iterator bio podržan u svim STL algoritmima, unutar naše iterator klase moramo imati definisane tipove:
- `iterator_category`
 - definiše tip iteratora korištenjem tag-klasa, npr.
`std::forward_iterator_tag`, `std::output_iterator_tag` ...
- `value_type`
 - definiše tip podatka koji se dobije dereferenciranjem iteratora,
- `difference_type`
 - mjera udaljenosti između dva iteratora. Koristi se kod random-access iteratora ali se mora definisati kao tip u svakom tipu,
- `pointer`
 - pokazivač na tip kojeg dereferencira iterator,
- `reference`
 - referenca na tip kojeg dereferencira pokazivač

Osobine iteratora - nastavak

- Primjer definicije osobina iteratora za klasu `iterator` definisanu u našoj klasi `Niz`:

public:

```
typedef std::bidirectional_iterator_tag iterator_category;
typedef Elem value_type;
typedef size_t difference_type;
typedef Elem* pointer;
typedef Elem& reference;
```

- Druga mogućnost (jednostavnija) je da klasa `iterator` nasljeđuje od odgovarajuće `std::iterator` klase u kojoj su definisani svi potrebni odgovarajući tipovi:

```
template<typename Elem>
class Niz<Elem>::iterator
    : public std::iterator<std::bidirectional_iterator_tag, Elem>
{
    // definicija članova i metoda
};
```